НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.272

«До захисту допущено»
Науковий керівник кафедри
_____ І.А. Дичка
«___»_____2018р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 121 Інженерія програмного забезпечення

на тему: «Модель реалізації крупнозернистого паралелізму в мультипроцесорних системах»

Виконав:
студент VI курсу, групи КП-61м
Гуров Владислав Олександрович _____

Керівник:
Доцент кафедри ПЗКС, к.т.н.,
Жабіна В.В. _____

Рецензент:
Доцент кафедри ОТ, к.т.н., доц.,
Верба О.А. _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2018 року

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Крупнозернистий паралелізм – паралелізм на рівні програм та програмних модулів.

Дрібнозернистий паралелізм – паралелізм на рівні операцій і мікрооперацій.

Потокові обчислювальні системи – системи, що використовують механізм управління обчисленнями, при якому команди виконуються, коли стають доступними їх операнди.

Система реального часу – це система, яка повинна реагувати на події у зовнішньому по відношенню до системи середовищі або впливати на середовище в межах необхідних тимчасових обмежень.

Мультипроцесорна система – це підклас багатопроцесорних комп'ютерних систем, де є декілька процесорів і один адресний простір, видимий для всіх процесорів.

ASIC (application-specific integrated circuit) – інтегральна схема, спеціалізована для вирішення конкретного завдання.

ПЛІС – програмована логічна інтегральна схема.

MSBI (master, slave, bus, interface) – функціональна графічна мова.

СЛАР – система лінійних алгебраїчних рівнянь.

WPF (Windows Presentation Foundation) – графічна (презентаційна) підсистема в складі .NET Framework 3.0.

XAML (eXtensible Application Markup Language) – розширювана мова розмітки для додатків.

ПЗ (програмне забезпечення) – сукупність програм системи обробки інформації і програмних документів, необхідних для експлуатації цих програм.

Діаграма прецедентів – це діаграма в UML, яка показує відношення між акторами та прецедентами в системі, що дозволяє описати систему на концептуальному рівні.

Діаграма послідовності – це діаграма в UML, яка відображає взаємодії об'єктів впорядкованих за часом та послідовність обміну повідомленнями між ними.

ВСТУП

Постійно зростаюча складність завдань реального часу, пов'язана з збільшенням області застосування систем управління. Сучасні потреби вимагають підвищення реакції систем. Розпаралелювання обчислень виникає на всіх рівнях обробки інформації. Безліч завдань реального часу вимагає швидкої реалізації крупнозернистих алгоритмів. Кричним є побудова систем, що дозволять ефективно реалізовувати крупнозернистий паралелізм і при цьому виявляти прихований паралелізм задач в динаміці.

Розпаралелювання обчислень зазвичай суттєво додає трудомісткості при розробці і налагодженні програми, адже це приходиться робити програмісту вручну. Ось чому в наш час дослідники намагаються створити методи та архітектури систем, що призведуть до спрощення створення і налагодження програм, що виконуються паралельно.

Для рішення задач, що являють собою крупнозернистий алгоритм використовую мультипроцесорні системи. В таких системах зазвичай один процесор виконує одне завдання крупнозернистого алгоритму.

Тому виникає потреба у досліджені можливості прискорення обробки інформації за рахунок динамічного розподілу робіт між обчислювальними вузлами в мультипроцесорних системах.

Для дослідження та вивчення характеристик мультипроцесорних систем при розв'язані крупнозернистих алгоритмів виникає потреба створення імітаційної системи.

Тому є доцільним розробка програмного забезпечення імітаційної системи. Для цього необхідно за допомогою засобів програмної інженерії створити архітектуру ПЗ та описати основні користувацькі та функціональні вимоги.

Розробити відповідне ПЗ на основі розробленої архітектура та відповідно до виокремлених вимог.

На основі створеної імітаційної моделі можна дослідити характеристики мультипроцесорної системи. Використовуючи отримані дані можна виявити найефективніші системи для рішення різних задач.

1. СУЧАСНИЙ СТАН ТА ПРОБЛЕМИ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ В СИСТЕМАХ РЕАЛЬНОГО ЧАСУ

1.1. Аналіз підходів роботи обчислювальних систем

Існує дві протилежні за загальним принципом роботи моделі обчислювальної системи, а саме модель обчислень, заснована на потоці команд управління, англомовна назва Controlflow, та модель обчислень, керована потоком даних, англомовна назва Dataflow.

Більшість сучасних обчислювальних машин, будь то суперкомп'ютер, звичайний персональний комп’ютер або навіть калькулятор, об'єднує загальна модель обчислень, заснована на потоці команд управління. Майже всі вони мають у своєму складі фон-нейманівську архітектуру. Тому часто архітектуру потоку команд управління називають фон-нейманівською, але вона є лише підмножиною архітектури потоку команд управління.

В рамках архітектури потоку команд управління обчислювальна машина складається з двох основних вузлів: процесора і пам'яті. Програма представляє собою набір інструкцій, що зберігаються в пам'яті в порядку виконання. Дані, з якими працює програма, також зберігаються в пам'яті у вигляді набору змінних. Адреса інструкції, яка виконується в даний момент, зберігається в спеціальному реєстрі. Момент початку виконання інструкції визначається моментом завершення попередньої [1].

Архітектура потоку команд управління має ряд вроджених вад, повністю позбутися від яких неможливо, так як вони виникають з самої організації обчислювального процесу, можна тільки зменшити негативний ефект з допомогою різних технічних рішень. Далі наведено основні проблеми:

- Перед виконанням інструкції її операнди необхідно завантажити з пам'яті в регістри процесора, а після виконання - вивантажити результат назад в пам'ять. Шина між процесором та пам'яттю стає вузьким місцем: процесор простоює частину часу, чекаючи завантаження даних.
- Побудова мультипроцесорних систем пов'язана з рядом труднощів. Існують дві основні концепції таких систем: із загальною і з розподіленою пам'яттю. У першому випадку складно фізично забезпечити спільний доступ багатьох процесорів до одного запам’ятовуючого пристрою. У другому випадку виникають проблеми когерентності даних і синхронізації. З ростом числа процесорів в системі все більше ресурсів витрачається на забезпечення синхронізації і все менше – на власне обчислення.
- Ніхто не гарантує, що на момент виконання будь-якої інструкції її операнди будуть знаходитися в пам'яті за вказаними адресами. Інструкція, яка повинна записати дані, може виявитися, що ще не виконалася. У багатопоточних застосуваннях істотна частка ресурсів витрачається на забезпечення синхронізації потоків.

У потокових машинах дані передаються і зберігаються у вигляді токенів. Токен – це структура, яка містить власне значення і мітку – покажчик вузла призначення. Найпростіша потокова обчислювальна система складається з двох пристроїв: виконавчого і пристрою формування.

На рис. 1 наведено структуру найпростішої потокової обчислюваної системи.

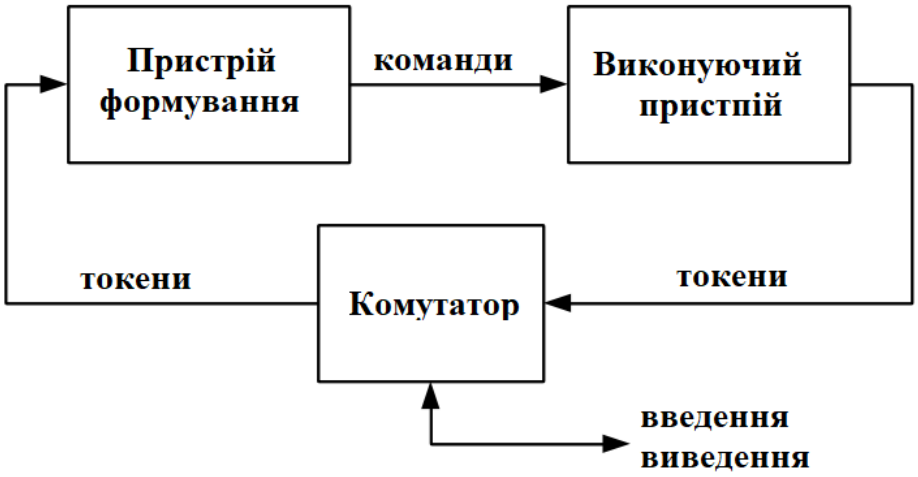


Рис. 1. Найпростіша потокова система

Виконуючий пристрій служить для виконання інструкцій і формування токенів з результатами операцій. Як правило, воно включає в себе пам'ять команд, доступну тільки для читання. Готовність вхідних даних вузла визначається за наявністю набору токенів з однаковими мітками. Для пошуку таких наборів і служить пристрій формування. Зазвичай вони реалізується на базі асоціативної пам'яті. Використовується або «справжня», апаратна асоціативна пам'ять (CAM - content-addressable memory), або структури, що працюють аналогічно, наприклад, хеш-таблиці.

Однією з основних перевагою потокової архітектури є її масштабованість: не складає труднощів зібрати систему, яка містить безліч пристроїв формування і виконуючих пристроїв. Пристрої об’єднуються найпростішим комутатором, причому для адресації токенів служать їх мітки. Весь діапазон номерів вузлів просто розподіляється рівномірно між пристроями. Ніяких додаткових заходів для синхронізації обчислювального процесу не потрібно [1].

Час рішення паралельних задач багато в чому визначається ефективністю максимального завантаження обчислювальних засобів за рахунок розпаралелювання обчислень. Для розробки паралельних програм створено велику кількість різних засобів програмування. Багато з них

ґрунтуються на принципово різних підходах, принципах, мають різну сферу застосування і ступінь поширення [2]. Наприклад, в монографії [3] наведено більше сотні найменувань засобів вирішення завдань на паралельних обчислювальних системах.

По одному з важливих для даної роботи ознак класифікації кошти паралельного програмування можна розділити на статичні і динамічні. Статичні засоби визначають прийнятний план виконання завдань по їх апріорним характеристикам, динамічні засоби модифікує план під час виконання завдань.

До найбільш широко використовуваним статичним засобів відносяться мови паралельного програмування (наприклад, ADA, C#, Java, CSP, C/C++), бібліотеки паралельного програмування (MPI, OpenMP, PVM), а також кошти, що надаються операційною системою (COM, CORBA) [4,5].

Статична розпаралелювання може виконуватися автоматично за допомогою спеціальних компіляторів, але їх можливості дуже обмежені [6]. Основні завдання розпаралелювання і синхронізації процесів в статиці вирішуються програмістом на етапі розробки програм. Загальний недолік засобів статичного розпаралелювання полягає в тому, що при аналізі алгоритмів не завжди вдається виявити прихований паралелізм завдання. В основному це зумовлено браком інформації про тривалість процесів в конкретному випадку, що залежить від особливостей системи.

Серйозні проблеми виникають при багатозадачному режимі функціонування систем, коли програми починають виконуватися і закінчуватися незалежно одна від одної. Забезпечити синхронізацію процесів, коли стратегія управління може вимагати асинхронного запуску різних програм навряд чи можливо.

Ще одним недоліком статичного розпаралелювання є часто виникає необхідність проводити розпаралелювання програми заново в разі зміни конфігурації системи, а також зміни розмірності вхідних даних.

Слід також зазначити, що зазначені статичні засоби орієнтовані на використання в мультипроцесорних та розподілених системах, в яких розпаралелювання доцільно проводити на рівні обробки програм і програмних модулів, оскільки на обмін даними витрачається багато часу. Розпаралелювання обчислень на рівні операцій в цьому випадку просто недоцільно.

Одним з перспективних підходів, що дозволяють усунути ряд недоліків статичної планування і спростити підготовку паралельних програм, є розробка засобів динамічного розпаралелювання процесів [7, 8].

Існує кілька напрямків реалізації динамічного розпаралелювання, до яких відносяться системи, керовані потоком даних, і системи з автотрансформації обчислювальної мережі. Об'єднуючою особливістю стратегій динамічного розпаралелювання є їх орієнтація на структурований паралелізм на основі моделі функціонального програмування. Цей факт можна пояснити тим, що функціональне програмування передбачає відсутність обміну даними між функціями через глобальні змінні, тому функції можуть виконуватися незалежно. Це властивість моделі функціонального програмування використовується засобами динамічного розпаралелювання для організації паралелізму на рівні незалежних функцій або завдань [15].

1.2. Характеристика алгоритмів управління в реальному часі

Вибір архітектури обчислювальної системи багато в чому визначається структурою реалізованих алгоритмів.

Як приклад завдань реального часу розглянемо завдання, які вирішуються в автоматичних системах управління, які пред'являють підвищені вимоги до швидкості перетворення даних [9, 10].

В автоматичних системах управління кожен цикл включає прийом даних з датчиків, обробку даних, формування керуючого впливу. Найбільш тривалим, як правило, є процес обробки даних. Наприклад, рішення задач управління вимагає відтворення заданих траєкторій (в тому

числі і в багатовимірному просторі), інтерполяції функцій, розрахунку контурної швидкості просування об'єкта, режимів сполучення тощо. При вирішенні зазначених завдань, як і завдань моделювання, використовуються чисельні методи лінійної алгебри, зокрема, для вирішення систем алгебраїчних і диференціальних рівнянь.

У роботах [11, 12] показано, що розглянуті алгоритми мають переважно послідовно-паралельну структуру. Графи алгоритмів містять паралельні і послідовні гілки. Характер обміну даними між гілками є переважно диференціальним. При диференціальному обміні повинна бути передбачена можливість передачі даних з однієї гілки в будь-яку іншу, тобто система повинна забезпечувати реалізацію повно зв’язних графів обміну даними між гілками алгоритмів. Стратегія управління в залежності від різних причин може змінюватися. Це вимагає використання різних алгоритмів обробки даних в різних циклах управління. Алгоритми обробки даних можуть також модифікуватися при адаптації до різних об'єктів управління, що характерно для проблемно-орієнтованих систем.

Для забезпечення високої швидкості обчислень, наприклад, при великому числі координат управління, використовують паралельні системи з різним рівнем розпаралелювання процесів [13, 14].

У паралельних обчислювальних системах перетворення інформації може здійснюватися одночасно в різних обчислювальних вузлах.

Як відомо, можна виділити кілька основних рівнів паралелізму:

- рівень окремих програм (алгоритмів);
- рівень програмних модулів (фрагментів алгоритмів);
- рівень команд (операцій);
- рівень мікрокоманд (мікрооперацій).

Алгоритми паралельної обробки інформації можна уявити у вигляді графа, вершинам якого відповідають функції перетворення даних, а дугам - вхідні дані і результати перетворення, передавання даних між функціями вершин графа.

Паралелізму на рівні програм і програмних модулів відповідає термін «крупнозернистий (coarse-grain) паралелізм», а на рівні команд і мікрокоманд – «дрібнозернистий (fine-grain) паралелізм». Відповідно до цього використовуються терміни «крупнозернистий алгоритм (граф)» і «дрібнозернистий алгоритм (граф)».

В даний час терміни «крупнозернистий паралелізм» і «дрібнозернистий паралелізм» широко використовуються при описі різних завдань. Хоча такий поділ алгоритмів досить умовно, воно дозволяє сформувати вимоги до систем, які реалізують розглянуті різновиди паралелізму.

Крупнозернистий паралелізм, як правило, реалізується в мультипроцесорних системах і розподілених паралельних системах. Такі системи містять процесори з програмним управлінням, комп'ютери, кластери, з'єднані певної комунікаційним середовищем, яка зазвичай реалізує повно зв'язний граф обміну даними. В один момент часу в різних обчислювальних вузлах системи можуть одночасно виконуватися різні програми і програмні модулі.

Дрібнозернистий паралелізм реалізується в системах, що містять більш прості обчислювальні засоби (процесорні елементи, обчислювачі). Термін "дрібнозернистий паралелізм" говорить про елементарності окремого обчислювального дії. Вершин дрібнозернистого графа відповідають окремі операції, функції (можливо, багатомісні), які можуть одночасно виконуватися в різних вузлах системи.

Ефективність паралельних обчислень залежить від реалізованого рівня паралелізму, що пов'язано з розміром зерна уявлення графа обчислень. Найбільш високий ступінь розпаралелювання може бути досягнута при поданні алгоритмів у вигляді графа з крупнозернистою структурою, коли вершин графа відповідають окремій підпрограмі. При цьому збільшується число паралельних гілок, що дає потенційну

можливість використовувати більшу кількість паралельно працюючих обчислювальних вузлів.

Швидкість обробки інформації пов'язана не тільки з мінімізацією часу виконання паралельних гілок, але і мінімізацією витрат на обмін інформацією між гілками. Виграш в швидкості обчислень при збільшенні ступеня розпаралелювання алгоритмів пов’язаний зі зростанням інтенсивності обміну між гілками. Цей фактор є дуже важливим і повинен враховуватися при виборі архітектури обчислювальної системи.

У зв'язку з вищевикладеним можна зробити наступні висновки.

Для ефективної реалізації даного класу крупнозернистих послідовно-паралельних алгоритмів в проблемно-орієнтованих системах доцільно забезпечити повний граф зв'язків між обчислювальними вузлами і високу швидкість реалізації алгоритмів. Необхідно передбачити можливість модифікації алгоритмів і доповнення нових алгоритмів в разі зміни характеристик об'єктів управління.

Крупнозернистий паралелізм має велике значення для систем моделювання. Для вирішення ряду завдань використовуються моделі крупнозернистих обчислень [15].

1.3. Сучасні інтерактивні системи проектування, налагодження та моделювання мультипроцесорних систем

Існує велика кількість організацій та компаній, які займаються розв’язанням вище описаних проблем та задач. Прикладами найбільших таких компаній є Xilinx та Altera.

Xilinx Inc. – американська компанія без власного виробництва, один з найбільших виробників програмованих логічних пристроїв. Заснована у 1984 році. В цьому ж році в компанії було винайдено, а в 1985 році почато виробництво першої комерційної мікросхеми FPGA. Компанія займається проектуванням напівпровідникових пристроїв програмованої логіки (FPGA, CPLD, ASIC, розробкою програмних модулів та бібліотек

інтелектуальної власності для програмування пристроїв програмованої логіки на мовах опису апаратних засобів (VHDL, Verilog), а також розробкою пакетів програмного забезпечення для програмування пристроїв програмованої логіки (пакет програм ISE Design Suite). Xilinx є підприємством без власних виробничих потужностей. Для виготовлення програмованих логічних пристроїв компанія співпрацює з різними виробниками інтегральних схем, такими як Samsung, TSMC або UMC [16].

Altera – один з найбільших розробників ASIC, програмованих логічних інтегральних схем (ПЛІС), була заснована в 1983 році. Як підприємство без власних виробничих потужностей, Altera концентрується в першу чергу на розробці схем і модулів на основі таких мов опису апаратури, як VHDL, Verilog і власний AHDL. В області виробництва мікросхем співпрацює з різними виробниками [17].

Основні вироби – це програмовані мікросхеми, а також послуги по перетворенню проектів під ПЛІС в ASIC для масового виробництва. Компанія також випускає програми для розробки програмно-апаратних засобів для ПЛІС, а також компілятори під ядро процесора власної розробки.

До переваг продуктів Altera відносять:

Енергоефективність – платформи Altera забезпечують покращену енергоефективність виконання робочих навантажень, що знижує загальне енергоспоживання та загальну вартість підтримки.

Гнучкість розгортання

Точність – обчислення можуть виконувати з будь-якою точністю та типом даних, від 64-бітної з плаваючою точкою до цілого числа до двійкового, надалі дозволяючи вам налаштувати вашу реалізацію відповідно до ваших точних потреб.

Швидкість – ПЛІС є швидкими. Коли важливий час відповіді, забезпечується достатня швидкість та низька затримка, що забезпечує швидше прийняття рішень

Типи вводу-виводу – пристрої вводу та виводу дозволяють здійснювати прямий доступ до будь-якого джерела даних та стандарту інтерфейсу, включаючи камери, пристрої для зберігання даних або мережу.

Майбутні алгоритми – оскільки ПЛІС перепрограмовуються, вони не тільки корисні для сьогоднішніх робочих навантажень та алгоритмів, але й адаптовані до архітектур майбутнього [18].

Продукти компаній Xilinx та Altera майже повністю задовольняють сучасним потребам та вимогам до обчислювальних систем. Але їх продукти дуже часто є універсальними, тому їх важка адаптувати та використовувати при рішеннях спеціалізованих вузько направлених задач з максимальною ефективністю використання ресурсів обчислювальних систем. Для досягнення цієї ефективності обчислювальні системи повинні максимально повністю враховувати специфіку області використання.

Також до недоліків можна віднести високу вартість продуктів Xilinx та Altera. Тому при вирішенні відносно невеликих задач використання дорогих продуктів може буди економічно невигідним.

Отже, враховуючи вище описане можна зробити висновок, що розробка невеликих архітектур потокових обчислювальних систем є актуальною задачею для вирішення потреб спеціалізованих вузько направлених областей.

1.4. Висновки до розділу 1

В даному розділі був проведений аналіз існуючих підходів роботи обчислювальних систем, що керуються потоком даних. Було проаналізовано їх переваги та недоліки.

Було розглянуто типи алгоритмів управління та виділено основні рівнів паралелізму: рівень окремих програм (алгоритмів), рівень програмних модулів (фрагментів алгоритмів), рівень команд (операцій), рівень мікрокоманд (мікрооперацій).

Дуло досліджено обчислення крупнозернистих алгоритмів.

Було розглянуто переваги та недоліки продуктів, які пропонують сучасні компанії для рішення проблем розпаралелювання.

На підставі проведеного аналізу були сформульовані завдання роботи, виконані в наступних розділах.

2. АРХІТЕКТУРА МУЛЬТИПРОЦЕСОРНОЇ СИСТЕМИ ТА КОНЦЕПЦІЯ ВИКОНАННЯ ОБЧИСЛЕНЬ

2.1. Модель обчислень крупнозернистих алгоритмів під управлінням даними

До складу мультипроцесорної системи входять процесорні модулі, які обмінюються даними через загальний адресний простір. При цьому кожен процесорний модуль може мати власну локальну пам'ять для незалежного виконання власної програми.

До перших проектів, пов'язаних з автоматичним розпаралелюванням обчислень, належать системи, що керуються потоком даних (*data-flow system*), які розроблялися для реалізації дрібнозернистого паралелізму [19-21]. Але такі системи неефективні при вирішенні задач, алгоритми яких мають грубозернисту структуру через велику кількість пересилань даних між вузлами системи.

Модель обчислень під управлінням даними використовується і для реалізації грубозернистих алгоритмів. У роботах [22-25] запропонована наступна концепція відповідної обробки даних.

1. Процес рішення задачі представляється за допомогою графа, кожній i-й вершині ($i = 1, .., m$) якого відповідає певний обсяг роботи (об-числень), а кожній дузі – потік даних, необхідних для виконання роботи. Граф розробляється без врахування кількості процесорних модулів.
2. Обчислення, що відповідають вершинам графа, є незалежними і взаємодіють між собою тільки через дані.

3. Для перетворення інформації створюється бібліотека підпрограм. Як компоненти бібліотеки можуть використовуватися програмні модулі, що функціонують в заданій операційному середовищі та дозволяють здійснити необхідне для вирішення задачі перетворення даних.

2.2. Формати дескрипторів завдань та даних

Вихідні дані для рішення задачі готуються на основі потокового графа у вигляді набору дескрипторів завдань і даних.

Дескриптор завдання для i-ї вершини графа має вигляд:

$$W = \{N_i, I_i, P_i, Q_i\}$$

- N_i – унікальне для глобальної задачі ім’я даного дескриптора завдання;
- I_i – ідентифікатор завдання;
- P_i – множина імен вихідних даних;
- Q_i – сумарне число вхідних потоків даних для i-го завдання.

Потік даних, що відповідає дузі графа між i-ю та j-ю вершинами, характеризується дескриптором даних:

$$D_{ij} = \{N_{ij}, n_{ij}, Q_{ij}, A_{ij}\}$$

- N_{ij} – унікальне для глобальної задачі ім’я даного дескриптора даних;
- n_{ij} – порядковий номер входження відповідного даного в вершину графа;
- Q_{ij} – сумарна кількість даних;
- A_{ij} – елемент адресації даних, що визначає місце даних у пам’яті системи.

З елементів дескрипторів формуються заявки на виконання i-го завдання:

$$Z_i = \{I_i, P_i, A_i\}$$

- I_i – ідентифікатор завдання;
- P_i – множина імен вихідних даних;
- A_i – множина всіх елементів адресації даних для виконання i-го завдання.

Для формування заявок Z_i створюється таблиця в пам'яті, яка доступна для всіх процесорів. При надходженні дескрипторів завдань у відповідні позиції рядків таблиці вводяться значення I_i та P_i. При надходженні дескрипторів даних у відповідні позиції рядків записуються елементи множини A_i.

У кожний рядок таблиці вводять значення лічильника C_i. При надходженні дескриптора порівнюють значення C_i та Q_i. Рівність цих значень є умовою активізації заявки.

Вільні процесори одержують заявки, а після виконання завдань повертають отриманий дескриптор даних, який бере участь при формуванні нової заявки. При цьому лічильник значення лічильника C_i встановлюється на нуль, тобто система готова для повторного формування заявки з такою назвою, якщо це необхідно.

Для моделювання роботи мультипроцесорної системи при обчисленні крупнозернистого графа, було побудовано граф на рис. 2 на основі наступної системи лінійних алгебраїчних рівнянь (СЛАР) (1).

$$\begin{cases} 8x_1 - x_2 + 9x_5 + 8x_6 + 5x_7 = 6 \\ 3x_1 + 9x_2 + x_3 + x_5 + 8x_6 + 5x_7 = 8 \\ 5x_1 + 11x_2 + 9x_5 + 8x_6 + 5x_7 = 10 \\ 2x_1 - x_2 + 13x_4 + x_5 + x_6 + 5x_7 = 5 \\ 6x_1 - 1x_2 + 9x_5 + 2x_6 + 5x_7 = -9 \\ 2x_1 + 10x_2 + 17x_3 + 9x_5 + 8x_6 + x_7 = 1 \\ 4x_1 - x_2 + 2x_4 + 15x_5 + 8x_6 + 5x_7 = 12 \end{cases} \quad (1)$$

В графі на рис. 2 використовуються наступні позначення:

- Ai – рядки матриці, що відповідає коефіцієнтам СЛАР (1);

- Di – рядки матриці, які є результатами відповідних обчислень підпрограм;
- $D35$ – вектор розв’язок СЛАР;
- номери в середині вершин графа задають, яку підпрограму потрібно виконати.

Підпрограма 1 описує одну операцію прямого ходу алгоритму Гауса для обчислення коренів системи лінійних рівнянь. Ця операція виконує віднімання двох рядків матриці так, щоб отримати 0 на початку другого рядка.

Підпрограма 2 описує одну операцію зворотного ходу алгоритму Гауса для обчислення коренів системи лінійних рівнянь. Ця операція виконує операцію знаходження одного кореня системи рівнянь.

Даний граф задається наступною матрицею (2).

$$
\begin{bmatrix}
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}
\quad (2)
$$

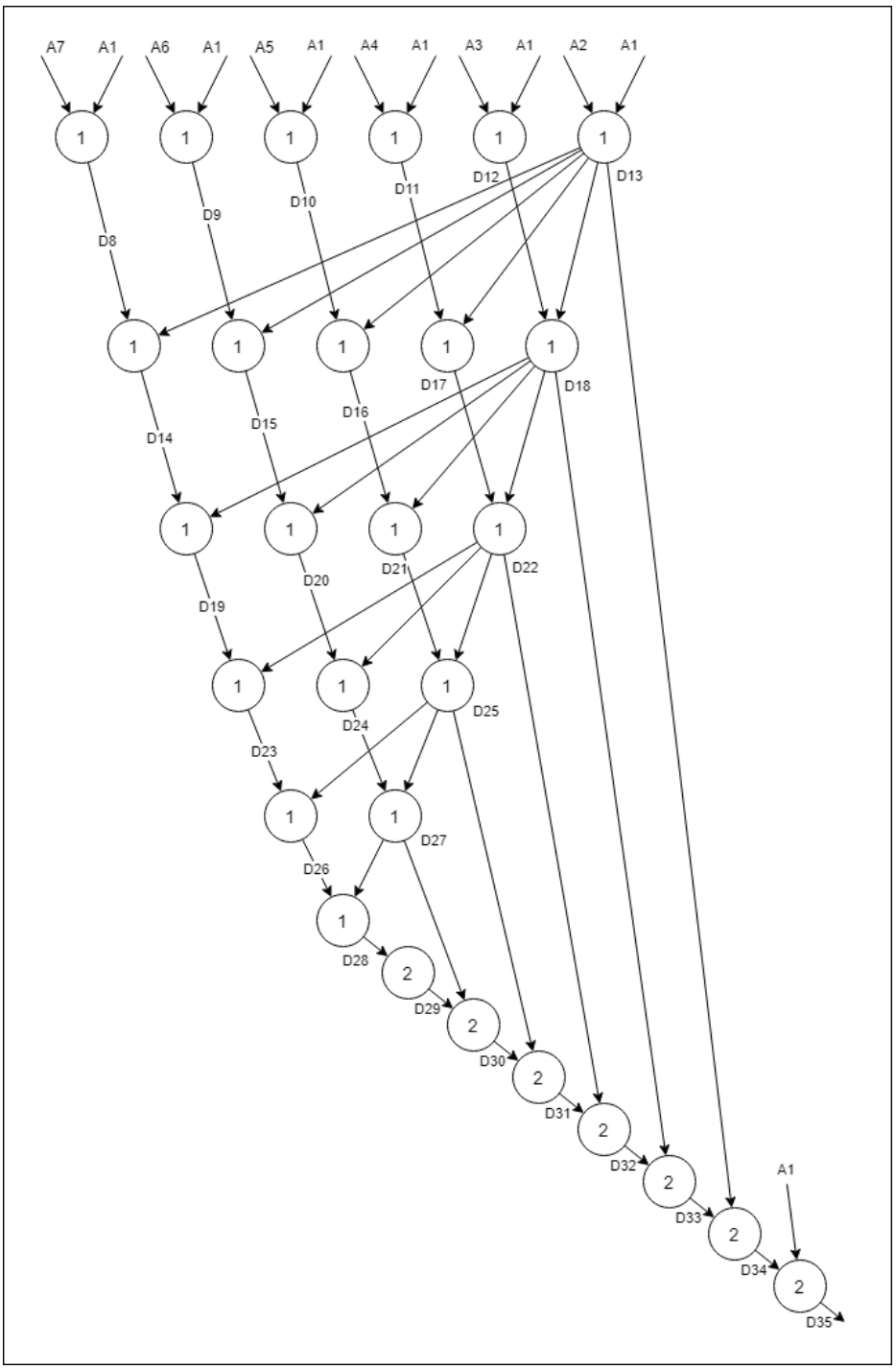


Рис. 2. Крупнозернистий граф розв'язання СЛАР

В результаті читання та обробки вхідних текстових файлів, отримаємо дескриптори завдань, що наведено в таблиці 1.

Таблиця 1. Список дескрипторів завдань для розв’язку СЛАР

Номер	Номер підпрограми	Вхідні дані	Кількість даних	Час
1	1	{ 8 }	16	48
2	1	{ 9 }	16	48
3	1	{ 10 }	16	48
4	1	{ 11 }	16	48
5	1	{ 12 }	16	48
6	1	{ 13-6 }	16	48
7	1	{ 14 }	14	42
8	1	{ 15 }	14	42
9	1	{ 16 }	14	42
10	1	{ 17 }	14	42
11	1	{ 18-5 }	14	42
12	1	{ 19 }	12	36
13	1	{ 20 }	12	36
14	1	{ 21 }	12	36
15	1	{ 22-4 }	12	36
16	1	{ 23 }	10	30
17	1	{ 24 }	10	30
18	1	{ 25-3 }	10	30
19	1	{ 26 }	8	24
20	1	{ 27-2 }	8	24
21	1	{ 28 }	6	18
22	2	{ 29 }	2	9
23	2	{ 30 }	4	18
24	2	{ 31 }	6	27
25	2	{ 32 }	8	36
26	2	{ 33 }	10	45
27	2	{ 34 }	12	54

2.3. Прискорення обробки даних

Прискорити обчислювальний процес можна за рахунок зменшення інтенсивності використання загальної шини при доступі процесорів до загального ресурсу, а також зменшення команд для синхронізації обміну даними між процесорами.

Застосування сучасної технологи «система на кристалі» дозволяє адаптувати структуру обчислювальних систем на ПЛІС до певних умов

експлуатації [26] (в тому числі, використовуючи неоднорідний доступ до пам'яті різних ПМ).

Пропонована архітектура системи представлена на рис. 3 з використанням функціональної графічної мови MSBI (Master, Slave, Bus, Interface) [27]. Для визначеності вважаємо, що процесор P_1 є управляючим процесорним модулем, а решта процесорів є підлеглими процесорними модулями [28].

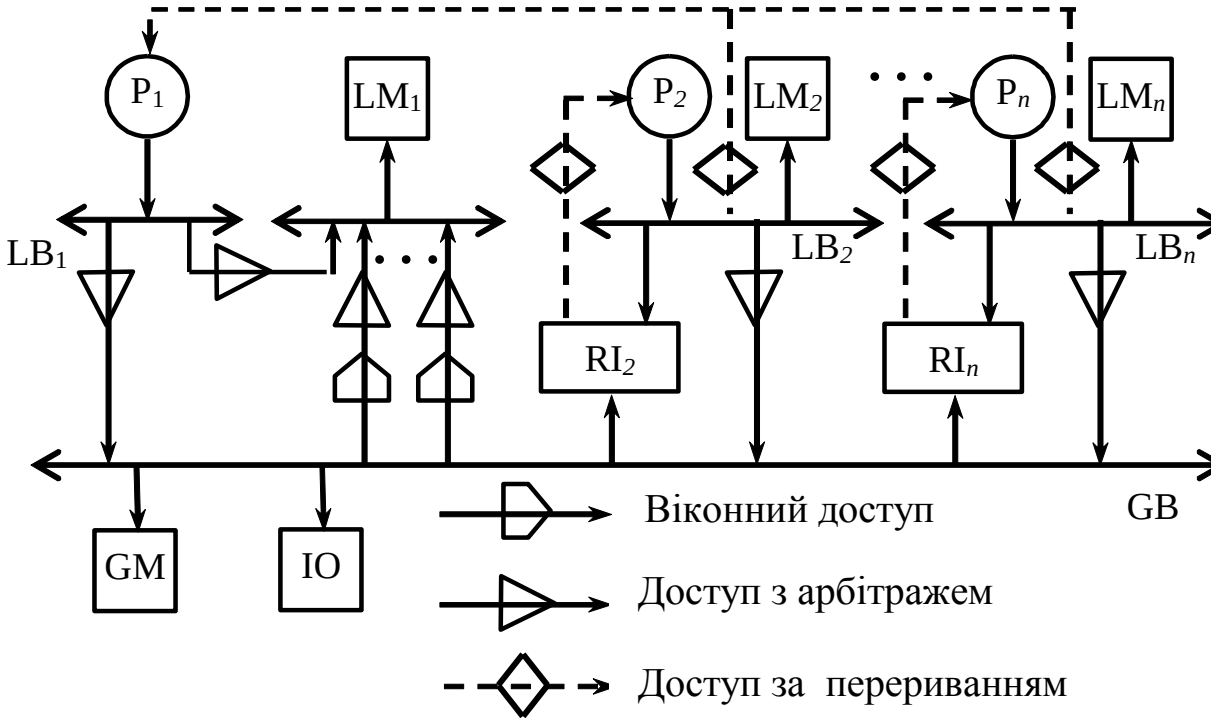


Рис. 3. Архітектура обчислювальної системи

На рис. 3 використано наступні позначення:

- P_i – процесор;
- LM_i – локальна пам'ять;
- GB_i – загальна пам'ять;
- IO_i – пристрої введення-виведення;
- RI_i – командний регістр;
- LB_i – локальна шина;

- GB_i – загальна шина.

Далі наведено основні принципи роботи пропонованої архітектури мультипроцесорної системи:

1. Управляючий процесор формує заявки для підлеглих процесорів, може відключати їх від загальної шини, задавати режим роботи через командний регістр КР.
2. Для скорочення часу очікування доступу до загального ресурсу в адресний простір управляючого процесорного модуля введена локальна пам'ять великого об’єму, в якій зберігається таблиця завдань і результати виконання обчислень.
3. Для кожного підлеглого процесора до цієї пам'яті організовано власний віконний доступ [25]. Це дозволяє працювати підлеглим процесорам з вікном без його захоплення семафором.
4. Для рівномірного розподілу завантаженості між обчислювальними процесорами було введено пріоритетне призначення завдання процесорам. Принцип якого полягає в тому, що чим більше часу процесор не виконує обчислення, тим більше стає його пріоритет. Процесор з найбільшим пріоритетом буде обчислювати наступне завдання. Після того, як процесору було призначене завдання, його пріоритет стає найменшим.

2.4. Алгоритм роботи мультипроцесорної системи

Управляючий процесор в трансляційному режимі записує в регістри RI_i команду, яка повідомляє про наявність готових завдань підлеглий процесорний модуль, що має максимальний динамічний пріоритет в даний момент часу, зчитує через своє вікно з локальної пам’яті управляючого процесора заявку і дані для неї. При читанні або запису масиву з M слів підлеглий процесор спочатку встановлює у вікні початкову адресу локальної пам'яті управляючого процесора, а потім виконує M звернень для пересилання даних, тобто практично при великих M для пересилання

одного слова здійснюється одне звернення до загальної шини. Про завершення звернень або виконання завдання підлеглий процесорний модуль повідомляє управляючий процесор через систему переривань [29].

При централізованому обміні даними через загальну пам'ять середнє число звернень на пересилку слова перевищує два [25]. Крім того, загальна шина використовується більш інтенсивно через необхідність захоплення вікон через семафори.

Для рівномірно розподілу завантаженості підлеглих процесорних модулів вводить поняття пріоритетного призначення завдання для виконання. Завдання для обчислення буде призначатися тому процесору, який має найбільший пріоритет. Пріоритет підлеглого процесора збільшується, якщо він не виконує ніяких обчислень. Після того як процесору була призначене завдання і він його виконав, його пріоритет встановлюється в мінімальне значення.

2.5. Висновки до розділу 2

У даному розділі була досліджена можливість підвищення ефективності обробки даних в мультипроцесорних системах під керівництвом потоків дескрипторів даних за рахунок динамічного розподілення робіт між процесорами.

Запропонована архітектура та модель реалізації системи, що дозволяє практично вдвічі зменшити кількість звернень до загального ресурсу при пересиланні даних, що дає потенційну можливість прискорити обробку інформації. Модель може бути реалізована за технологією «система на кристалі» з використанням сучасної елементної бази. Одержані результати дають змогу розробити програмну модель системи для дослідження режимів обробки інформації.

Таким чином, отримані результати підтверджують ефективність застосування моделі обчислень під керуванням дескрипторів даних в мультипроцесорних системах при реалізації крупнозернистого паралелізму.

3. РОЗРОБКА МОДЕЛІ МУЛЬТИПРОЦЕСОРНОЇ СИСТЕМИ

Розроблювальна система представляє собою настільний додаток, що призначений для моделювання обчислювальних процесів крупнозернистого паралелізму в мультипроцесорних системах. Основною задачею системи є те, що вона дозволяє користувачу імітувати процеси паралельного обчислення задачі, яка відповідає вимогам крупнозернистого алгоритму. Також вона містить модуль моніторингу стану системи в певний момент часу.

Для розробки імітаційної системи було сформовано вимоги, щодо функціональних можливостей програмного забезпечення.

3.1. Вимоги до імітаційної системи

3.1.1. Список термінів і визначень

Користувач – особа, яка аналізує імітаційний процес виконання паралельних обчислень крупнозернистого алгоритму в мультипроцесорній системі.

Мультипроцесорна система – це система, яка складається з декількох процесорів та має один спільний адресний простір, до якого мають доступ всі процесори [30].

Крупнозернистий алгоритм – граф задач, в якому вершина графа відповідає деякій підпрограмі, а дуга – відповідає даним, які пересилаються між підпрограмами.

Підпрограма – це фрагмент програми, що реалізує деякий алгоритм обчислення, що має на вході необхідну множину вхідних даний та генерує на виході множину вихідних даних, які можуть пересилатися для виконання іншої підпрограми.

3.1.2. Загальні вимоги

Призначення системи – надання функціональних можливостей для моделювання процесу виконання паралельних обчислень крупнозернистого алгоритму в мультипроцесорних системах.

Система надає наступний спектр можливостей пов'язаних з:

- налаштуванням основних складових частин мультипроцесорної системи;
- завантаженням вхідних даних, які задають крупнозернистий алгоритм;
- виконанням паралельних обчислень в заданій мультипроцесорній системі з заданими вхідними даними;
- моніторингом стану мультипроцесорної системи в певний момент виконання обчислень.

Простота інтерфейсу та гнучкість системи дозволяють користувачу користуватися нею з самого початку [31].

3.1.3. Вимоги користувача

Розроблювальна система надає наступні функціональні можливості для користувача:

- користувач може задавати кількість підлеглих процесорних модулів на початку роботи з системою;
- користувач може задавати вхідні данні;
- користувач може контролювати хід виконання паралельних обчислень;
- користувач може отримувати інформацію про стан системи в певний момент час виконання обчислень;
- користувач може отримувати проміжні та кінцевий результат виконання обчислень [32].

3.1.4. Специфікація функціональних вимог

1. Система повинна виконувати імітацію обчислень крупнозернистого алгоритму в мультипроцесорній системі. Відповідно до заданої архітектури мультипроцесорної системи та заданого алгоритму виконання паралельних обчислень.
2. Система дає можливість змінювати кількість підлеглих процесорних модулів.
    2.1. Система повинна надавати можливість змінювати кількість процесорів на початку роботи. Після запуску процесу обчислення, користувач не повинен мати змогу додавати чи видаляти обчислювальні модулі.
    2.2. При натисканні кнопки «Додати обчислювальний процесор» в імітаційні системі додається новий процесорний модуль. Максимальна кількість процесорний модулів повинна буди обмежена певною величиною.
    2.3. При натисканні кнопки «Видалити обчислювальний процесор» з імітаційної системи прибирається один процесорний модуль. Мінімальна кількість процесорний модулів повинна дорівнювати одному модулю.
    2.4. Всі зміни елементів імітаційної системи повинні відображатися на інтерфейсі користувача.
3. Система повинна надавати можливість задавати вхідні данні через текстовий файл.
    3.1. Користувач повинен задати в текстовому файлі вхідну матрицю, яка задає граф крупнозернистого алгоритму. Номери рядків та стовпців вхідної матриці повинні відповідати номерам вершин графа. Цифра 1 у заданому рядку та стовпцю вказує на те, що між відповідними вершинами є зв’язок, причому це зв’язок з вершини номер, якої відповідає номеру рядку, до вершини, номер, якої

відповідає номеру стовпця. Цифра 0 у заданому рядку та стовпцю вказує на те, що між відповідними вершинами немає зв’язку.

3.2. Користувач повинен задати в текстовому файлі номери підпрограми кожній вершині, яка повинна виконувати обчислення в відповідній вершині.

3.3. Користувач повинен задати в текстовому файлі вхідні дані, які необхідні для обчислення заданого алгоритму.

3.4. Система повинна зчитувати задані користувачем текстові файлі, та на основі них виконувати обчислення.

4. Система повинна надавати можливість задавати підпрограми, які повинні виконуватися на підлеглих процесорних модулях. Також, ці підпрограми повинні відповідати номерам в вершинах крупнозернистого графа.

5. Система повинна відображати стан елементів імітаційної системи в кожен момент часу.

5.1. Система повинна показувати інформація про стан управляючого процесорного модуля, а саме таблицю, що відображає пам’ять, в якій зберігаються проміжні результати, таблицю в якій зберігаються команди, що очікують вхідні данні. Крім цього, система повинна надавати інформацію про кількість даних, які вже готові для кожної команди, та про загальну кількість даних, які потрібні для активізації команди.

5.2. Система повинна показувати інформація про стан підлеглих процесорних модулів. Якщо процесорний модуль не виконує підпрограму, то система повинна показувати, що процесор пустий і готовий для отримання нової задачі. Під час виконання підпрограми процесором, система повинна відображати таблицю, що відповідає вхідним даним,

інформацію, яка містить в регістрі процесора. Також потрібно відображати інформацію про поточний такт виконання підпрограми та загальну кількість тактів, які необхідно виконати.

5.3. Система повинна відображати окремим елементом інформацію про завантаженість підлеглих процесорних модулів.

5.4. Система повинна відображати зв’язки між управляючим та підлеглими процесорними модулями.

5.5. Система повинна відображати загальну кількість виконаних тактів процесу паралельного обчислення графа в кожен момент часу.

6. Система повинна надавати можливість користувачу управляти часом під виконання обчислень, тобто задавати виконання такту.

6.1. При натисканні кнопки «Такт» в імітаційні системі виконується один такт обчислень.

6.2. При натисканні кнопки «Розв’язати» в імітаційні системі автоматично виконуються всі такти, які обхідні для завершення обчислень, тобто для отримання кінцевого результату [33, 34].

3.1.5. Можливості системи

Для зручного та наочного представлення можливостей імітаційної системи виконання паралельних обчислень крупнозернистого алгоритму в мультипроцесорних системах було використано діаграму прецедентів та діаграми послідовності.

В ході аналізу системи було виявлено одного актора – «Користувач», що має зв’язок с наступними прецедентами:

- введення вхідної інформації, який включає в себе наступні прецеденти:
  - задання вхідного графу крупнозернистого алгоритму;

- задання вхідних даних;
- задання підпрограм;

- моніторинг стану системи в певний момент час виконання обчислень;
- управління тактами, який складається з двох наступних можливих варіантів:
  - виконання одного такту обчислень;
  - виконання всіх тактів до отримання кінцевого результату обчислень;
- задання кількості підлеглих процесорних модулів, який складається з двох наступних можливих варіантів:
  - видалення одного процесора;
  - додавання одного процесора [35].

Діаграма прецедентів представлена рис. 4.

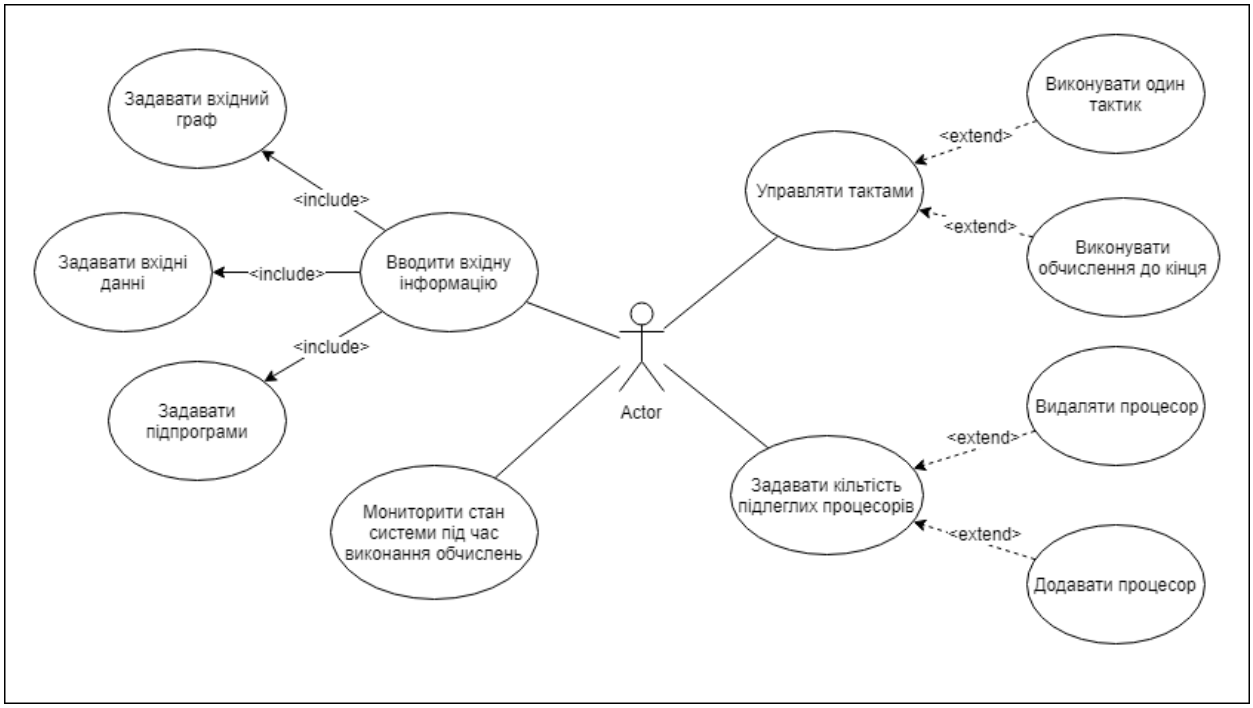


Рис. 4. Діаграма прецедентів

В ході побудови архітектури імітаційної системи виконання паралельних обчислень крупнозернистого алгоритму в мультипроцесорних

системах було створено дві діаграми послідовності, що охоплюють основні задачі даної системи:

- налаштування елементів мультипроцесорної системи;
- процес виконання паралельних обчислень.

Діаграма послідовності налаштування елементів мультипроцесорної системи представлена на рис. 5 [36].

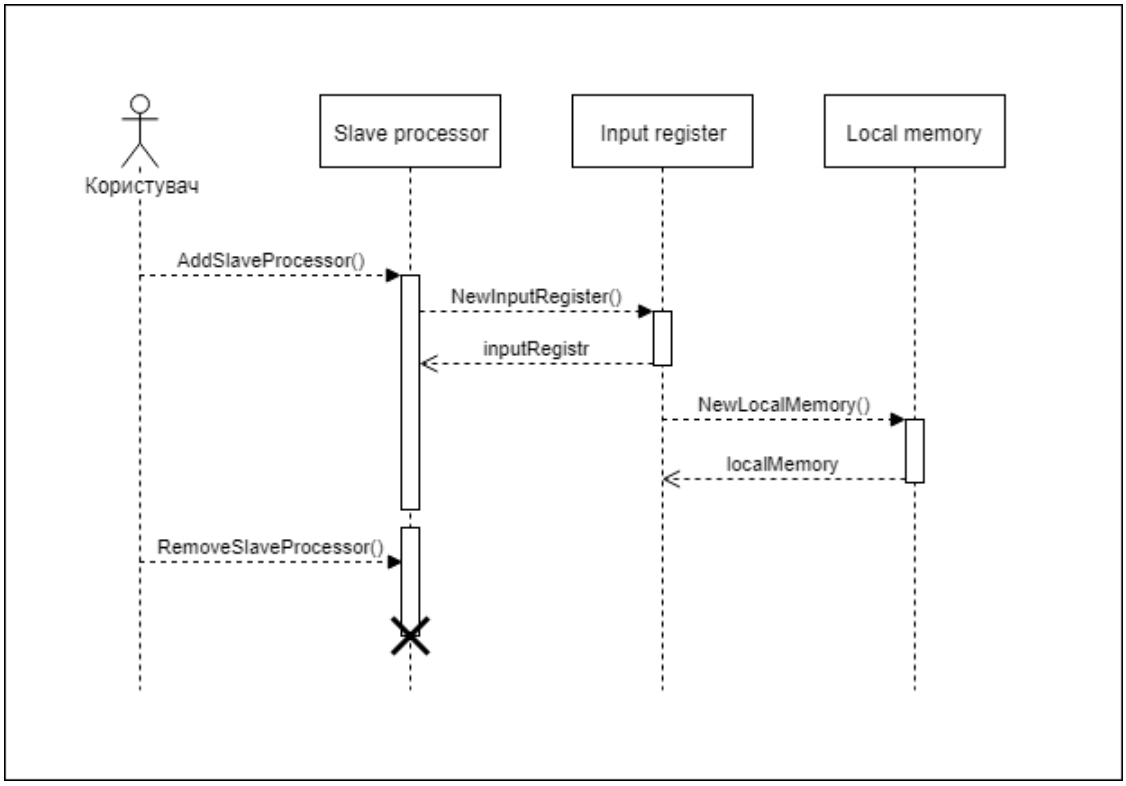


Рис. 5. Діаграма послідовності налаштування елементів мультипроцесорної системи

Ця діаграма описує процес додавання та видалення підлеглих процесорних модулів. При додаванні процесорного моду створюється відповідний об’єкт, до якого строються та додається об’єкт, що відповідає за роботу вхідного регістра процесора. Також строються та додається об’єкт, який імітує локальну пам’ять процесора. При видалені підлеглого процесорного модуля, видаляється один з процесорів мультипроцесорної системи [37].

Діаграма послідовності процесу виконання обчислень в мультипроцесорній системі представлена на рис. 6.

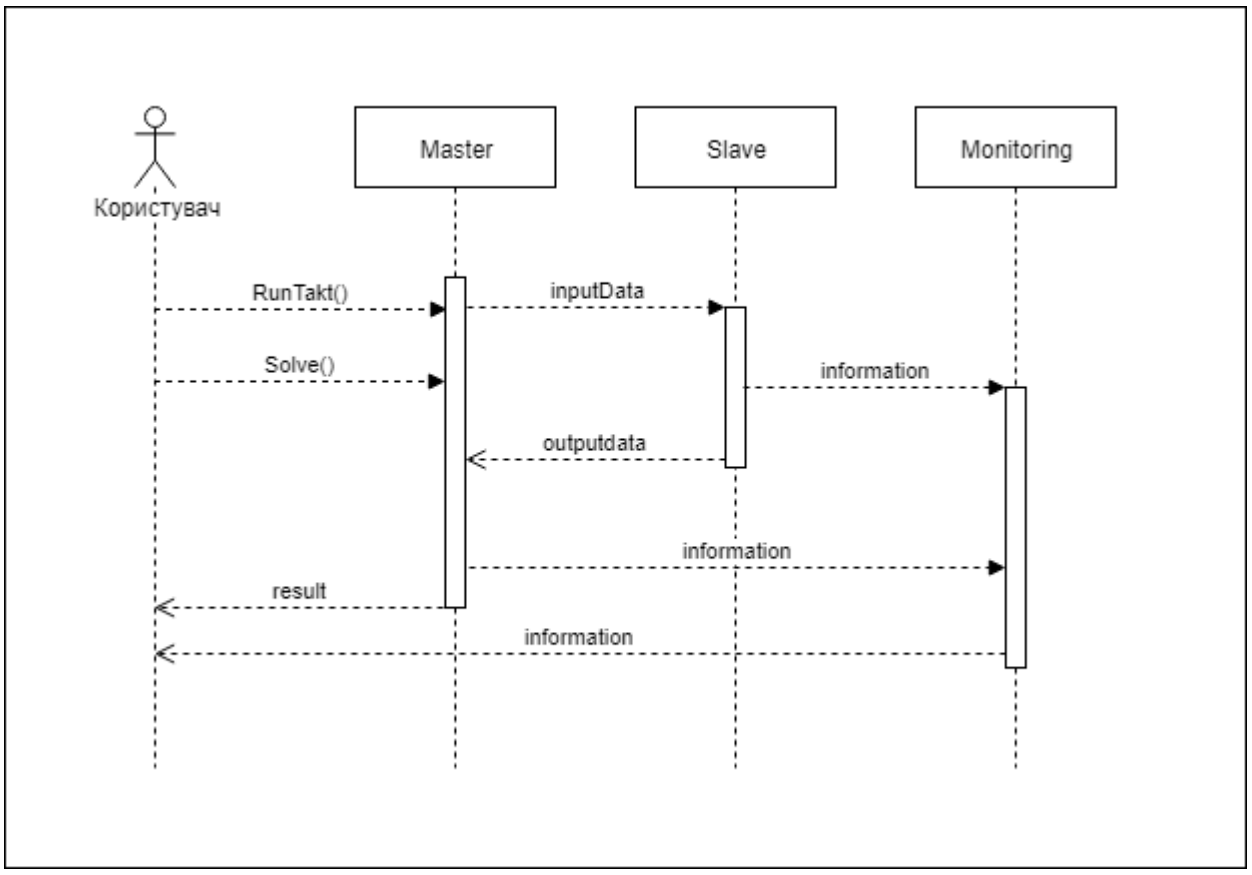


Рис. 6. Діаграма послідовності процесу виконання обчислень

Діаграма послідовності виконання обчислень описує загальний процес роботи імітаційної моделі мультипроцесорної системи. Користувач може задавати управляючому процесору режим роботи, а саме по тактом або автоматичний режим. Між управляючим та підлеглими процесорами відбувається обмін даними, який відповідає процедурі обчислення крупнозернистого алгоритму в мультипроцесорній системі. При цьому модуль моніторингу збирає інформацію про стани процесорних модулів та надає користувачу необхідну інформацію для аналізу процесу обчислення в досліджуваній системі [38].

3.1.6. Архітектура системи

Загальна структура архітектури імітаційної системи виконання паралельних обчислень крупнозернистого алгоритму в мультипроцесорних системах складається з наступних частин:

- інтерфейс користувача – модуль системи, через який користувач отримує інформацію про процес обчислення. Також, через

інтерфейс користувач може задавати режим роботи імітаційної системи, тобто по тактам або автоматичний режим;

- емулятор системи – модуль, який безпосередньо моделює процес паралельного обчислення крупнозернистого алгоритму в мультипроцесорній системі;
- блок моніторингу – модуль, який збирає інформацію про стани процесорних модулів та надає цю інформацію користувачу через інтерфейс;
- блок роботи з текстовими файлами – модуль, який працює з текстовими файлами, через які користувач задає крупнозернистий граф та вхідні данні. Даний модуль зчитує дані з текстових файлів та передає їх до модуля, який будує граф;
- інтерпретатор графу – модуль, який будує граф з наданої інформації. Також даний модуль створює дескриптори завдань та дескриптори даних, для тих даних, які користувач задав у файли з вхідними даними.

Архітектури імітаційної системи наведено на рис. 7.

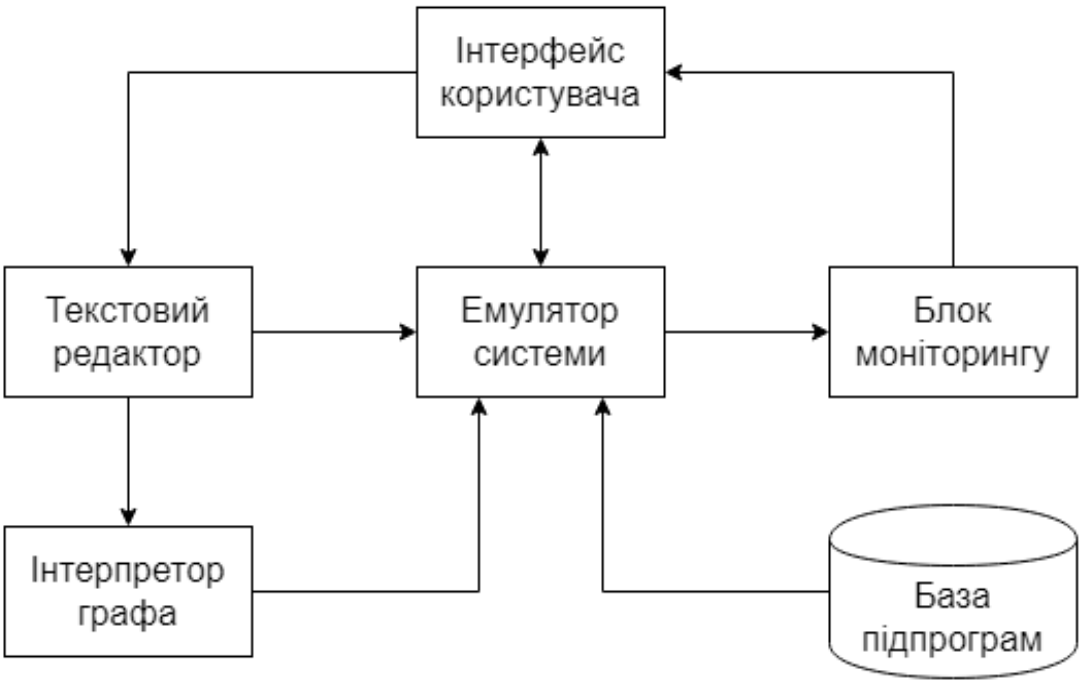


Рис. 7. Архітектура системи

3.2. Розробка класів імітаційної системи

Дескриптор завдання та дескриптор даних описують класи *Job* та *Data* відповідно.

Клас *Job*, який наведено на рис. 8, має наступні основні поля:

- *int Name* – ідентифікатор дескриптора завдання, який є унікальним для загальної задачі;
- *int TaskIdentifier* – ідентифікатор завдання, тобто номер підпрограми, яку необхідно виконати;
- *List<int> ListInputDataIdentifiers* – множина всіх вхідних імен дескрипторів даних;
- *int CountInputData* – сумарне число вхідних потоків даних для виконання завдання;
- *int Time* – додаткове поле, через яке можна задавати час виконання завдання.

```
class Job
{
    0 references
    public int Name { get; set; }
    1 reference
    public int TaskIdentifier { get; set; }
    2 references
    public List<int> ListInputDataIdentifiers { get; set; }
    1 reference
    public int CountInputData { get; set; }
    1 reference
    public int Time { get; set; }

}
```

Рис. 8. Клас *Job*

Клас *Data*, який наведено на рис. 9, складається з таких полів:

- *int Name* – ідентифікатор дескриптора даних, який є унікальним для загальної задачі;
- *int ArcIndex* – порядковий номер входження відповідної дуги в вершину графа
- *int CountData* – сумарне число даних;

- *int DataAddress* – елемент адресації даних, що визначає місце даних у пам’яті системи.

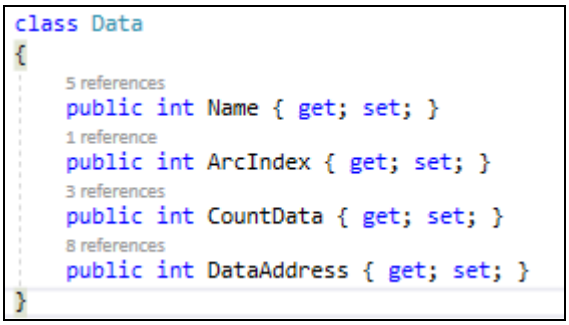


Рис. 9. Клас *Data*

Заявка описується класом *Request*, який наведено на рис. 10, складається з наступних основних полів:

- *int TaskIdentifier* – ідентифікатор завдання, тобто номер підпрограми, яку необхідно виконати;
- *List<int> ListInputDataIdentifiers* – множина всіх вхідних імен дескрипторів даних;
- *List<int> ListDataAddresses* – множина всіх елементів адресації даних для завдання.

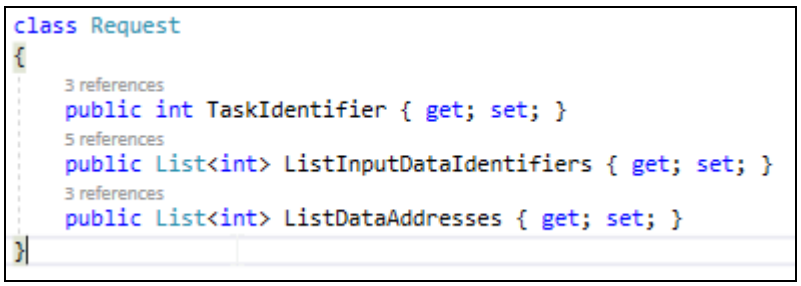


Рис. 10. Клас *Request*

Класи *MasterProcessor* та *SlaveProcessor* описують основні елементи мультипроцесорної системи, а саме управляючий процесорний модуль та підлеглий процесорній модуль відповідно.

До складу класу *MasterProcessor*, який наведено на рис. 11, входять наступні основні поля та методу:

- *List<CreateRequestElement> ListRequests* – таблиця, в якій зберігаються заявки, які очікують надходження необхідних дескрипторів даних;

- *Dictionary<int, double> ListResults* – таблиця, що імітує пам’ять управляючого процесорного модуля, в якій зберігаються проміжні результати алгоритму, обчисленні підлеглими процесорними модулями;
- *List<SlaveProcessor> ListSlaveProcessors* – список, в якому зберігається всі доступні в імітаційній системі підлеглі процесорні модулі;
- *int GlobalTime* – поле, в якому зберігається значення, що відповідає загальній кількості виконаних тактів.
- метод *bool InsertNewJob()* – метод, який додає нові дескриптори завдань в управляючий процесор. При надходженні дескриптора завдання, створюється нова заявка, яка додається до списку *ListRequests;*
- метод *bool InsertData()* – метод, який додає нові дескриптори даних в управляючий процесор. При надходженні дескриптора даних, в таблиці *ListRequests* заполюються відповідні поля, також заповнюється таблиця *ListResults;*
- метод *bool RunTakt()* – метод, який відповідає за виконання одного такту в управляючому процесорному модулі;
- метод *bool ActivationRequest()* – метод, який активізує заявку, яка отримала всі необхідні дескриптори даних, тобто повідомляє вільний підлеглий процесорній модулю, по доступність підпрограми для виконання;
- метод *bool ReadData()* – метод, який дозволяє організувати читання даних;
- метод *bool WriteData()* – метод, який дозволяє організувати запис даних;
- метод *bool AddSlave()* – метод, який додає один підлеглий процесорний модуль в систему;

- метод *bool RemoveSlave()* – метод, який видаляє один підлеглий процесорний модуль з системи.

```
class MasterProcessor
{
    7 references
    public List<CreateRequestElement> ListRequests { get; set; }
    9 references
    public Dictionary<int, double?> ListResults { get; set; }
    7 references
    public List<SlaveProcessor> ListSlaveProcessors { get; set; }

    public int GlobalTime;
    1 reference
    public MasterProcessor()...
    2 references
    public bool InsertNewJob(Job newJob)...
    2 references
    public bool InsertData(Data newData)...
    2 references
    public bool RunTakt()...
    1 reference
    public bool ActivationRequest(CreateRequestElement ActivationedRequest)...
}
```

Рис. 11. Клас *MasterProcessor*

Клас *SlaveProcessor,* який наведено на рис. 12, складається з таких елементів:

- *Dictionary<int, double> ListInputData* – таблиця, що імітує пам’ять підлеглого процесорного модуля, в якій зберігаються вхідні дані для виконання підпрограми;
- *Dictionary<int, List<double>> ListOutputData* – таблиця, в якій зберігаються вихідні результати отриманні після виконання підпрограми;
- *Registr InputRegistr* – поле, яке відповідає за роботу вхідного регістра підлеглого процесорного модуля;
- *bool FlagRunTask* – поле, яке описує стан підлеглого процесорного модуля. Значення *true* цього поля означає, що процесор знаходить у стані виконання підпрограми;
- *bool FlagSloveTask* – поле, яке описує стан підлеглого процесорного модуля. Значення *true* цього поля означає, що процесор виконав обчислення підпрограми;

- *int TimeLoadData* – поле, в якому зберігається значення кількості тактів, які потрібно виконати для читання всіх необхідних вхідних даних, для виконання підпрограми, з пам’яті управляючого процесорного модуля;
- *int TimeRunTask* – поле, в якому зберігається значення кількості тактів, які потрібно виконати для виконання підпрограми підлеглим процесорним модулем;
- *int CurrentTakt* – поле, в якому зберігається поточне значення виконаних тактів;
- метод *bool LoadTask()* – метод, який відповідає за завантаження необхідної підпрограми для виконання підлеглим процесорним модулем;
- метод *bool RunTakt()* – метод, який відповідає за виконання одного такту в підлеглому процесорному модулі;
- метод *bool Finalization()* – метод, який відповідає за завершення виконання підпрограми та формування всіх вихідних дескрипторів даних.

```
class SlaveProcessor
{
    6 references
    public Dictionary<int, double?> ListInputData { get; set; }
    5 references
    public Dictionary<int, List<double>> ListOutputData { get; set; }
    14 references
    public Request InputRegistr { get; set; }
    4 references
    public bool FlagRunTask { get; set; }
    4 references
    public bool FlagSloveTask { get; set; }
    6 references
    public int TimeLoadData { get; set; }
    4 references
    public int TimeRunTask { get; set; }
    10 references
    public int CurrentTakt { get; set; }
    1 reference
    public bool LoadTask()...
    1 reference
    public bool RunTakt()...
    1 reference
    public bool Finalization()...
}
```

Рис. 12. Клас *SlaveProcessor*

Елемент мультипроцесорної системи, який імітує пристрій введення та виведення описаний класом *InputOutput*, що наведено на рис. 12, який складається з наступних основних полів та методів:

- *List<Job> ListJobs* – поле, в якому знаходяться всі дескриптори завдань, які необхідно завантажити в управляючий процесорний модуль;
- *List<Tuple<Data, List<double>>> ListData* – поле, в якому знаходяться всі зовнішні, тобто початкові, дескриптори даних та самі значення даних, які необхідно завантажити в управляючий процесорний модуль;
- метод *Job InputJob()* – метод, який дозволяє організувати процес завантаження дескриптора завдання з пристрою введення-виведення в управляючий процесор;
- метод *Tuple<Data, List<double>> InputData()* – метод, який дозволяє організувати процес завантаження дескриптора даних та сами значень даних з пристрою введення-виведення в управляючий процесор.

```
class InputOutput
{
    public List<Job> ListJobs;
    public List<Tuple<Data, List<double>>> ListData;
    1 reference
    public InputOutput()...
    2 references
    public Job InputJob()...
    2 references
    public Tuple<Data, List<double>> InputData()...
}
```

Рис. 13. Клас *InputOutput*

Приклади підпрограм, які можуть виконувати підлеглі процесорні модулі, наведено на лістингу 1 та лістингу 2.

Підпрограма *GaussProgram1*, що наведена на лістингу 1, описує одну операцію прямого ходу алгоритму Гауса для обчислення коренів системи лінійних рівнянь. Ця операція виконує віднімання двох рядків матриці так, щоб отримати 0 на початку другого рядка.

Лістинг 1. Операція прямого ходу Гауса

```
    static List<double> GaussProgram1(List<double> data)
    {
        List<double> a1 = data.GetRange(0, data.Count / 2);
        List<double> a2 = data.GetRange(data.Count / 2, data.Count / 2);
        double coef = a1[0] / a2[0];
        return a1.Zip(a2, (f, s) => f - s * coef).ToList().GetRange(1, a1.Count -
1);
    }
```

Підпрограма *GaussReturnProgram1*, що наведена на лістингу 2, описує одну операцію зворотного ходу алгоритму Гауса для обчислення коренів системи лінійних рівнянь. Ця операція виконує операцію знаходження одного кореня системи рівнянь.

Лістинг 2. Операція знаходження одного кореня системи рівнянь

```
    static List<double> GaussReturnProgram1(List<double> data)
    {
        List<double> a1 = data.GetRange(0, data.Count / 2);
        List<double> a2 = data.GetRange(data.Count / 2, data.Count / 2);

        double sum = 0;
        for (int i = 1; i < a1.Count; i++)
        {
            sum += a1[i] * a2[i];
        }

        a2[0] = (a2[0] - sum) / a1[0];

        return a2;
    }
```

3.3. Побудова інтерфейсів імітаційної системи

Загальний інтерфейс імітаційної системи паралельних обчислень крупнозернистого графа в мультипроцесорній системі наведено на рис. 14.

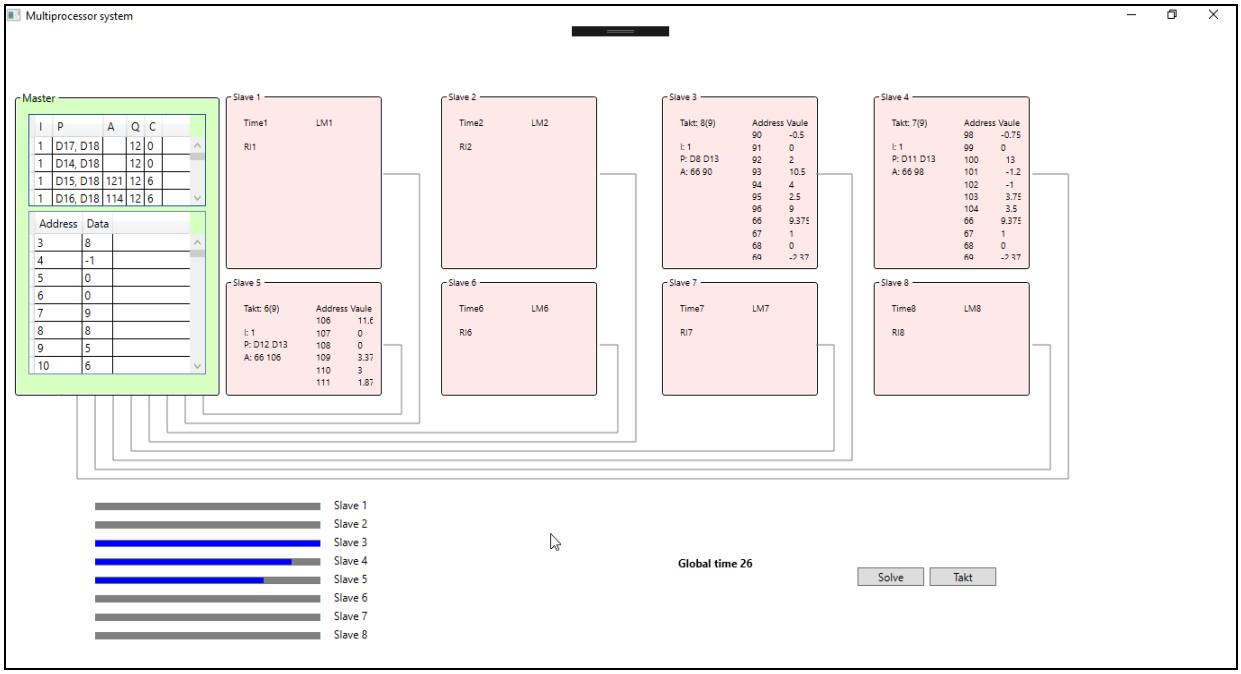


Рис. 14. Головне вікно системи

Інтерфейс відображення стану управляючого процесорного модуля наведено на рис. 15. В якому верхня таблиця відображає таблицю заявок, а нижня відображає пам’ять управляючого процесора, в якій зберігаються проміжні результати обчислень. Опис полів таблиці заявок наведено далі:

- *I* – показує номер підпрограми, яку буде виконувати підлеглий процесор;
- *P* – показує, які дескриптори даних потрібні для виконання підпрограми;
- *A* – показує адреси даних вже готових для обчислення дескрипторів даних;
- *Q* – показує, яка потрібна загальну кількість даних;
- *C* – показує загальну кількість даних, які вже надійшли до управляючого процесора.

В таблиці пам’яті проміжних результатів поле *Address* показує адресу даних в пам’яті, а поле *Data* показує саме значення даного.

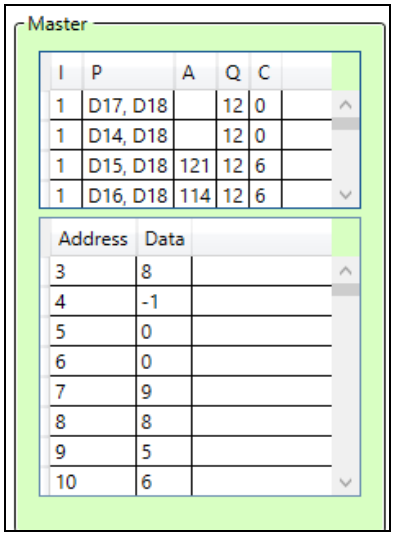


Рис. 15. Інтерфейс управляючого процесора

Інтерфейс відображення стану підлеглого процесорного модуля наведено на рис. 16. Зверху зазначається номер підлеглого процесора. Поле *Takt* показує поточний такт та в дужках загальний кількість тактів, які потрібно виконати. Поля *I*, *P*, *A* показують ідентифікатор підпрограми, вхідні дескриптори даних та адреси даних в пам’яті відповідно. В таблиці локальної пам’яті процесора поле *Address* показує адресу даних в пам’яті, а поле *Data* показує саме значення даного.

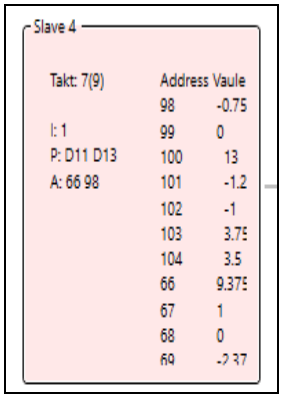


Рис. 16. Інтерфейс підлеглого процесора

Інтерфейс управління системою представлено на рис. 17. Він складається з поля *Global time*, в якому відображається загальна кількість виконаних тактів обчислення системою. Таж він має 4 кнопки, які відповідають за наступне:

- *Add Slave* – дозволяє користувачу додати один підлеглий процесорний модуль в систему;
- *Del Slave* – дозволяє видалити один підлеглий процесорний модуль із системи;
- *Solve* – кнопка, яка запускає режим автоматичного виконання обчислень в системі;
- *Takt* – дозволяє користувачу виконати один такт обчислення в імітаційній моделі.

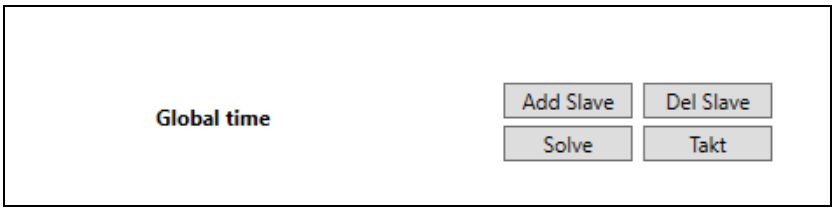


Рис. 17. Інтерфейс управління системою

Інтерфейс, який дозволяє користувачу переглядати поточну завантаженість підлеглих процесорних модулів наведено на рис. 18. Даний інтерфейс складається з слуг завантаженості кожного підлеглого процесора.

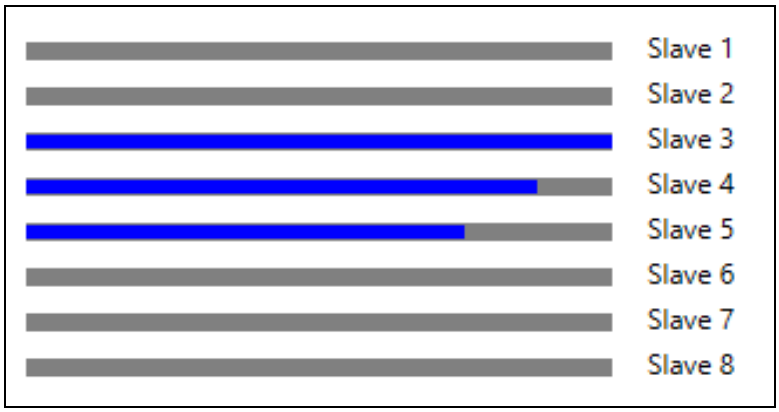


Рис. 18. Інтерфейс перегляду завантаженості підлеглих процесорних модулів

3.4. Висновки до розділу 3

В даному розділі були описані основні моменти процесу розробки імітаційної системи обчислення крупнозернистих графів в мультипроцесорній системі.

Були сформовані вимоги до відповідної системи та описано їх специфікації. Також було виділено можливості імітаційної системи.

Було розроблено та описано архітектуру системи.

Також було розроблено основні класи програмного забезпечення та інтерфейси користувача.

4. АНАЛІЗ ПРОДУКТИВНОСТІ ІМІТАЦІЙНОЇ МОДЕЛІ МУЛЬТИПРОЦЕСОРНОЇ СИСТЕМИ

Для оцінки продуктивності моделі реалізації крупнозернистого паралелізму в мультипроцесорних системах було розроблено імітаційну систему, та проведено моделювання паралельних обчислень на основі розробленої системи. Моделювання дає можливість порівняти продуктивність обчислювальної системи залежно від наступних параметрів:

- кількість підлеглих процесорних модулів;
- різні крупнозернисті графи.

Також було проведено порівняння ефективності мультипроцесорної системи з системою, яка виконує обчислення без використання паралелізму, тобто послідовне.

4.1. Результати моделювання при розв’язанні СЛАР

В таблиці 2 наведені результати обчислення графа на рис. 2, що задає крупнозернистий алгоритм розв’язання СЛАР. Моделювання обчислень відбувалося при різних кількостях підлеглих процесорних модулів, від 1-го до 8-ми процесорів.

Таблиця 2. Результати моделювання розв'язку СЛАР

Кількість ППМ	1	2	3	4	5	6	7	8
Час виконання (тактів)	1402	876	714	659	656	598	598	598

На рис. 19 наведено відповідну часу діаграму обчислень.

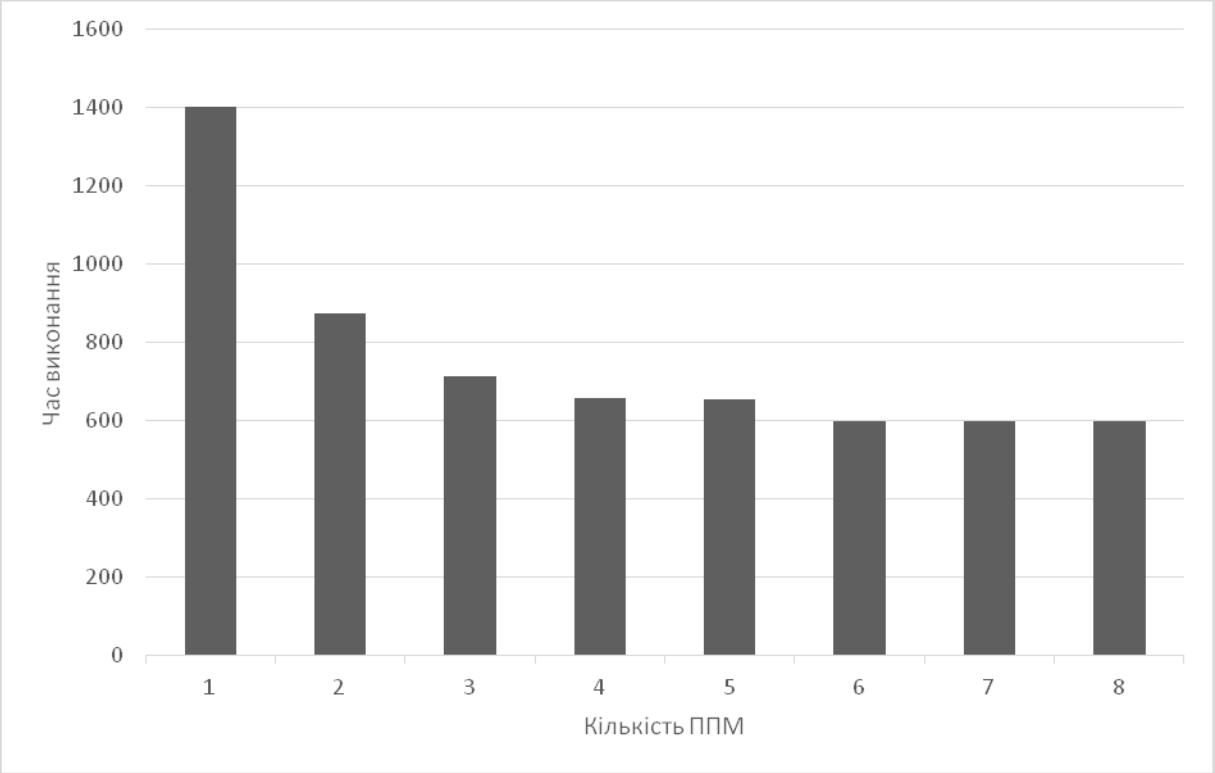


Рис. 19. Часова діаграма виконання обчислень при розв'язанні СЛАР

Отже, за таблицею 2 та графіком на рис.17 видно, що найшвидше обчислення виконуються при 6 підлеглих процесорних модулів. При цьому збільшення кількості підлеглих процесорів не впливає на час отримання результату.

Найефективніша мультипроцесорна система з шістьма підлеглими процесорами пояснюється тим, що ярусно-паралельний граф має шість потоків. Тому збільшення кількості обчислювальних процесорів не покращують результат, а їх зменшення призводить до збільшення час обчислень.

В таблиці 3 наведено результати, що показують, який відсоток від загального часу розв’язання СЛАР конкретний обчислювальний процесор був завантажений, тобто виконував обчислення або зчитував дані з таблиці проміжних результатів управляючого процесорного модуля. Наведені в таблиці 3 дані були отримані в імітаційній моделі без пріоритетного призначення задачі підлеглому процесору, загальний принцип, якого: чим

більше часу процесор не виконує обчислень, тим більше стає його пріоритет для призначення наступного готового завдання.

Таблиця 3. Результати завантаженості процесорів при розв'язку СЛАР без пріоритетного призначення

Кількість ППМ									Середня завантаженість ППМ
1	97%								97%
2	97%	59%							78%
3	96%	50%	44%						63%
4	96%	46%	40%	25%					51%
5	96%	36%	31%	25%	18%				41%
6	95%	40%	34%	28%	20%	10%			37%
7	95%	40%	34%	28%	20%	10%	0%		32%
8	95%	40%	34%	28%	20%	10%	0%	0%	28%
Номер ППМ	1	2	3	4	5	6	7	8	

Як видно з таблиці 3, найбільше обчислень виконує підлеглий процесор з найменшим порядковим номером. Це пояснюється тим, що в умовному списку процесорів він знаходить ближче всього. Тому, як тільки він завершує обчислення, йому буде призначена наступне готове для обчислення завдання.

Вище зазначеним причинами можна пояснити, чому кожен умовно ступний процесор виконує все менше обчислень та умовно останній має найменшу завантаженість.

Також з таблиці 3 видно, що при надлишковій кількості обчислювальних процесорів всі додаткові процесори зовсім не виконують обчислень і їх продуктивність дорівнює 0.

Для розв’язання отриманих проблем, тобто для рівномірного розподілу завантаженості процесорів, пропонується використати пріоритетне призначення готового завдання процесору.

Результати, що показують, який відсоток від загального часу розв’язання СЛАР конкретний підлеглий процесорний модуль був завантажений при пріоритетному призначенню завдання для обчислення, наведено в таблиці 4.

Таблиця 4. Результати завантаженості процесорів при розв'язку СЛАР з пріоритетним призначенням завдання

Кількість ППМ									Середня завантаженість ППМ
1	97%								97%
2	77%	79%							78%
3	61%	63%	67%						63%
4	53%	54%	50%	50%					51%
5	38%	44%	43%	46%	35%				41%
6	40%	42%	43%	41%	30%	30%			37%
7	29%	30%	32%	34%	33%	35%	34%		32%
8	34%	35%	36%	36%	22%	20%	21%	21%	28%
Номер ППМ	1	2	3	4	5	6	7	8	

Середні значення завантаженості обчислювальних процесорів за результатами двох моделювань для різної кількості процесорів однакові. Також однаковими будуть значення часу виконання обчислень. При цьому розподілення завантаженості між підлеглими процесорами відбувається більш рівномірно, що видно з таблиці 4.

Збереження тенденції до деякого зменшення завантаженості обчислювальних процесорів з більшими умовними номерами пояснюється наступними причинами.

1. Невеликою кількість завдань для обчислення. При значному збільшені кількості вершин графа завантаженість процесорів буде більш рівномірно розподілятися.
2. Перші готові для обчислення завдання призначаються умовно першим в списку обчислювальним процесорам. Для вирішення цієї проблеми можна використати випадкове призначення завдання обчислювальному процесору при рівному значенню пріоритетів двох та більше процесорів. Але при використанні в реальних умовах кількість завдань для обчислення значно збільшується, це призводить до мінімізації різниці завантаженості між обчислювальними процесорами.

В таблиці 5 наведені результати моделювання розв’язку СЛАР без паралелізму. Тобто в системі існує лише 1 процесор, який виконує функції управляючого та підлеглих процесорів. Час розв’язання СЛАР майже в півтора рази менше, ніж при використанні 1 управляючого процесора і 1 підлеглого, але при цьому більше, ніж при використанні 1 управляючого процесора і 2 підлеглих.

Таблиця 5. Результати моделювання розв'язку СЛАР без паралелізму обчислень

Тип системи	**Час виконання обчислень**
Обчислення виконує управляючий процесор	1080
1 управляючий процесор і 1 підлеглий	1402
1 управляючий процесор і 2 підлеглих	876

Такі результати пов’язані із затримкою при пересиланні даних від управляючого процесора до підлеглого. В системі з одним процесором такі пересилання не виконуються.

Також при збільшені кількості даних, які потрібні для виконання підпрограм, та якщо сам час виконання підпрограм буде не значний,

ефективність використання даної мультипроцесорної системи буде зменшуватися. Але подібні збільшення кількості даних та зменшення часу виконання підпрограм в графі після певної границі будуть суперечити поняттю крупнозернистого графа.

4.2. Результати моделювання при операціях з матрицями

Для дослідження параметрів системи при обчислені різних крупнозернистих графів в мультипроцесорній системі було використано алгоритм обчислення виразу (3) з чотирьох квадратних матриць розмірністю 3 на 3.

$$\mathbf{A \times B + C^2 + D \times (B^T + A \times D)} \quad (3)$$

На основі виразу (3) було побудовано крупнозернистий граф, який зображено на рис. 20. Номери в середині вершин графа означають, яку підпрограму необхідно виконати:

- 1 – транспонування матриці;
- 2 – сума двох матриць;
- 3 – множення двох матриць;
- 4 – взведення матриці в квадрат, тобто множення матриці саму на себе;
- 5 – сума трьох матриць.

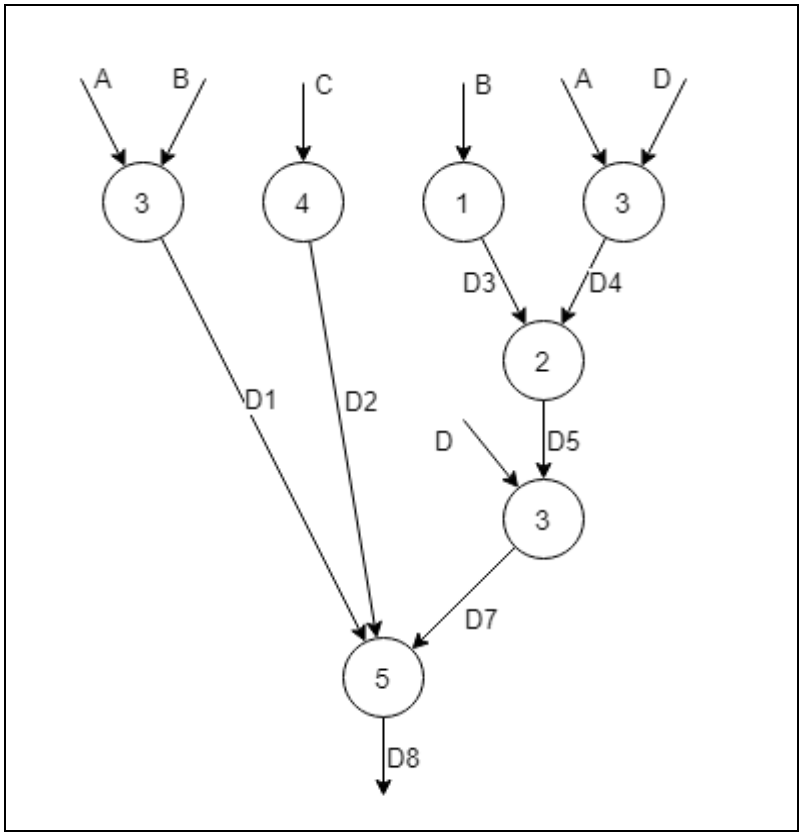


Рис. 20. Крупнозернистий граф обчислення виразу з квадратних матриць

Крупнозернистий Граф, що зображений на рис. 18, задається наступною матрицею (4).

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (4)$$

Таблиця 6. Список дескрипторів завдань для обчислення виразу з матриць

Номер	Номер підпрограми	Вихідні дані	Кількість даних	Час
1	3	{ 1 }	18	90
2	4	{ 2 }	9	90
3	1	{ 3 }	9	10
4	3	{ 4 }	18	90
5	2	{ 5 }	18	20
6	3	{ 7 }	18	90
7	5	{ 8 }	27	30

В результаті читання та обробки вхідних текстових файлів, отримаємо дескриптори завдань, що наведено в таблиці 6.

В таблиці 7 наведені результати обчислення заданого графа. Моделювання обчислень відбувалося при різних кількостях підлеглих процесорних модулів, від 1-го до 8-ми процесорів.

Таблиця 7. Результати моделювання обчислення виразу з матриць

Кількість ППМ	1	2	3	4	5	6	7	8
Час виконання (тактів)	546	418	329	319	319	319	319	319

На рис. 19 наведено відповідну часу діаграму обчислень.

За таблицею 7 та графіком на рис. 21 видно, що найшвидше обчислення виконуються при 4 підлеглих процесорних модулів. При цьому збільшення кількості підлеглих процесорів не впливає на час отримання результату.

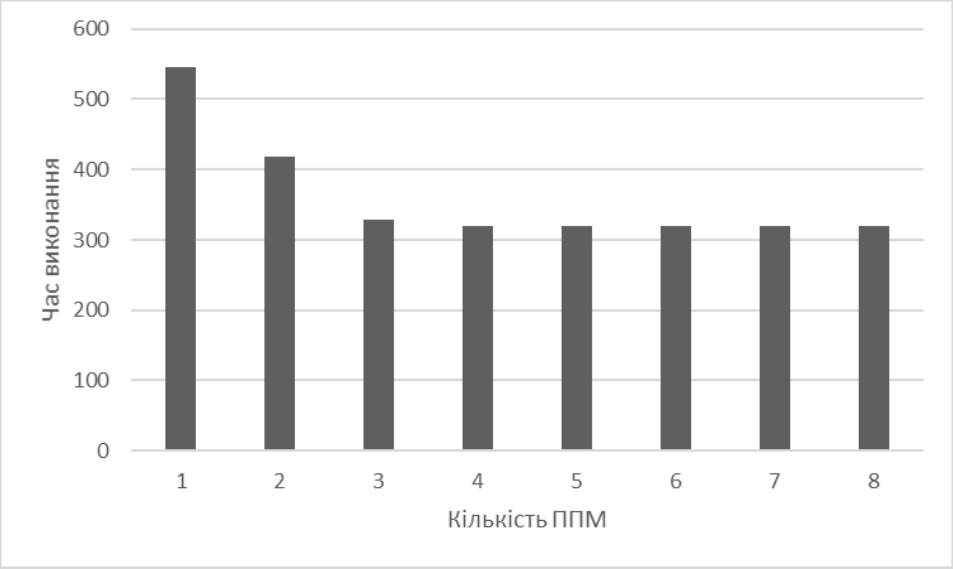


Рис. 21. Часова діаграма обчислення виразу з матриць

Наведенні вище результати були отримані при обчисленні звичайних матриць, тобто не одиничних, не нульових, не діагональних тощо.

Припустимо, що матриці **в** та **с** є діагональними або одиничними. Тоді мікропрограми в вершинах 1 та 2 будуть виконувати швидше. В таблиці 8 наведені результати обчислення при заданих умовах.

Таблиця 8. Результати моделювання обчислення виразу з матриць з умовами

Кількість ППМ	1	2	3	4	5	6	7	8
Час виконання (тактів)	406	348	319	319	319	319	319	319

За результатами моделювання, які наведені в таблиці 8, видно, що найшвидше обчислення виконуються при 3 підлеглих процесорних модулів. Тобто на основі графу видно, що є 4 потоки, найшвидше обчислення буде при використанні чотирьох процесорів. При обчисленні одним процесором найдовшої гілки, два інших процесора виконали обчислення трьох потоків. Це пов’язано з тим, що час обчислення головної гілки графа значно більше ніж час виконання побічних гілок. Даний

приклад показує перевагу методу розпаралелювання на рівні потоків даних.

На рис. 22 зображено граф, в якому вершина з підпрограмою, що виконує суму трьох матриці, розбита на дві вершини з підпрограмами, що виконують суму двох матриць. Тобто загальну суму було розбито на дві.

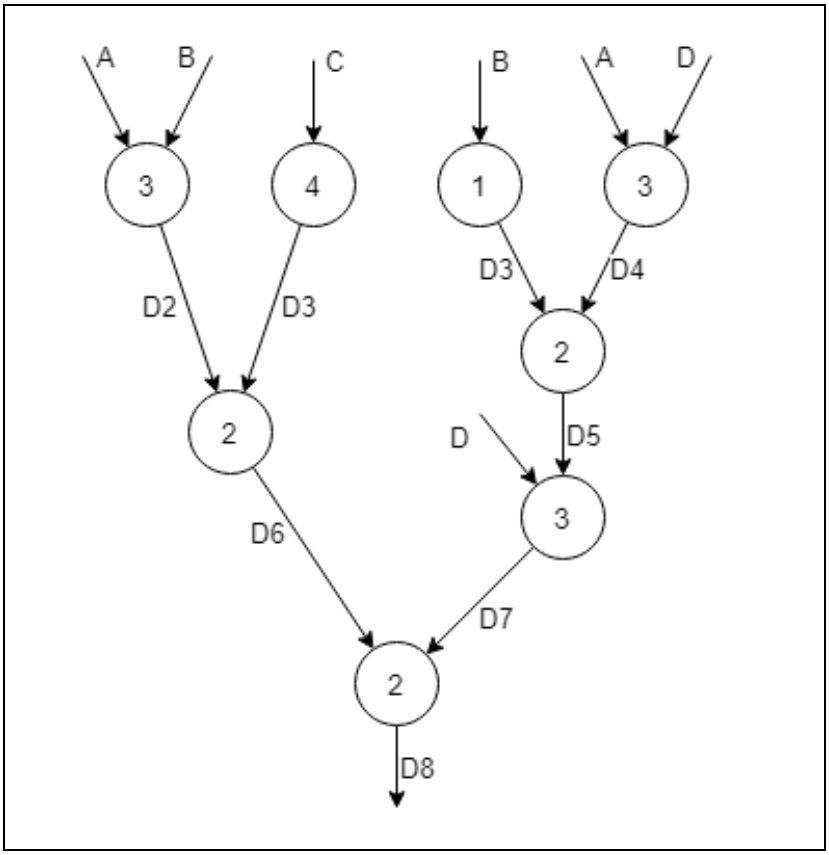


Рис. 22. Крупнозернистий граф обчислення виразу з квадратних матриць з розбитою сумою

В таблиці 9 наведені результати обчислення граф обчислення виразу матриць з розбитою сумою.

Таблиця 9. Результати моделювання обчислення виразу з матриць з розбитою сумою

Кількість ППМ	1	2	3	4	5	6	7	8
Час виконання (тактів)	426	329	300	300	300	300	300	300

За результатами моделювання видно, що швидкість обчислення зросла для кількості підлеглих процесорних модулів більше одного.

Але такі розбиття не завжди можуть призвести до покращення ефективності обчислень. Адже при показаному розбиті дві вершини в сумі виконувалися приблизно на чверть часу більше, ніж одно з сумою трьох матриць. Також було приблизно на чверть більше пересилань даних.

Результати трьох моделювань зведені в таблиці 10.

Таблиця 10. Результати трьох моделювань обчислення виразу з матриць

Кількість ППМ	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Зі Звичайними матрицями	546	418	329	319	319	319	319	319
З діагональними матрицями	406	348	319	319	319	319	319	319
З діагональними матрицями та розбитою сумою	426	329	300	300	300	300	300	300

На рис. 23 наведено графіки отриманих результатів трьох моделювань обчислення виразу з квадратних матриць.

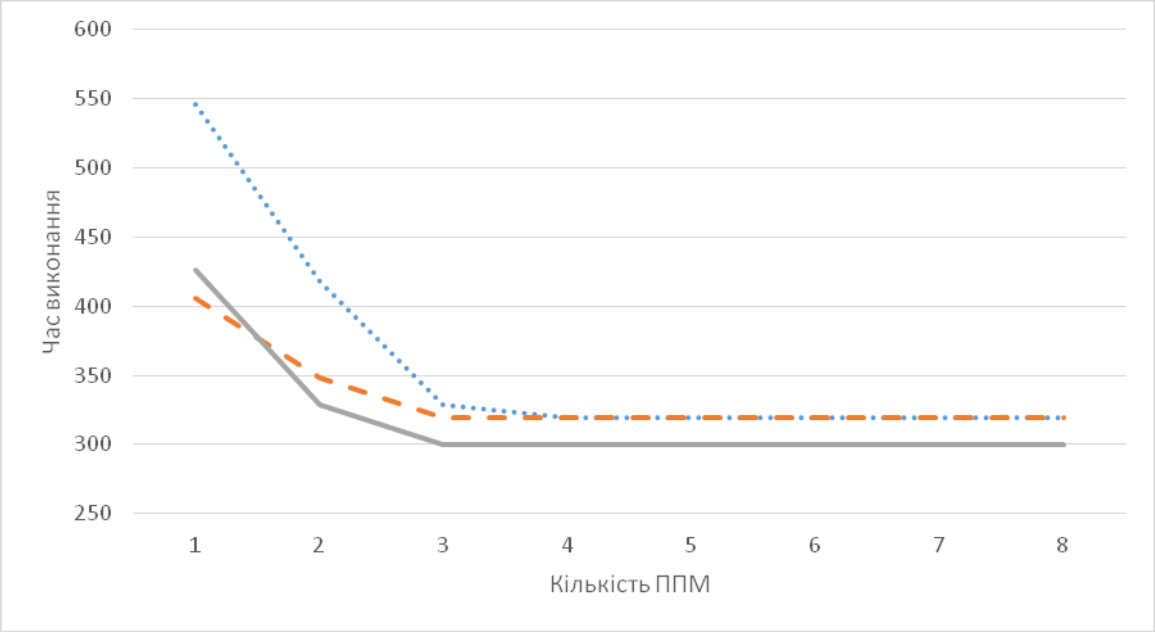


Рис. 23. Результати трьох моделювань обчислення виразу з матриць

Для дослідження впливу прихованого паралелізму при обчислені крупнозернистих графів в мультипроцесорній системі було використано алгоритм обчислення виразу (4) з 3 квадратних матриць розмірністю 3 на 3.

$$\mathbf{A} \times \mathbf{C}^{\mathrm{T}} + (\mathbf{A}^2 + \mathbf{B}^2 + \mathbf{C}^{\mathrm{T}}) + (\mathbf{C}^{\mathrm{T}})^2 \quad (4)$$

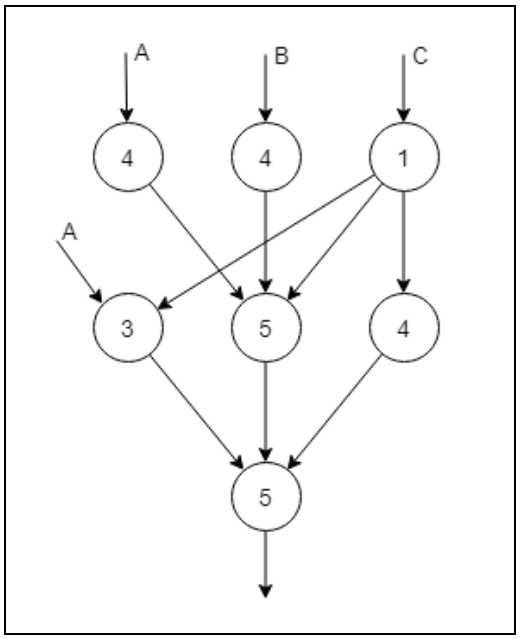


Рис. 24. Крупнозернистий граф виразу з 3 квадратних матриць

В таблиці 11 наведені результати обчислення заданого графа. Моделювання обчислень відбувалося при різних кількостях підлеглих процесорних модулів, від 1-го до 8-ми процесорів.

Таблиця 11. Результати моделювання обчислення виразу з 3 матриць

Кількість ППМ	1	2	3	4	5	6	7	8
Час виконання (тактів)	546	338	259	218	218	218	218	218

На рис. 25 наведено відповідну часу діаграму обчислень.

За таблицею 11 та графіком на рис. 25 видно, що найшвидше обчислення виконуються при 4 підлеглих процесорних модулів. При цьому збільшення кількості підлеглих процесорів не впливає на час отримання результату.

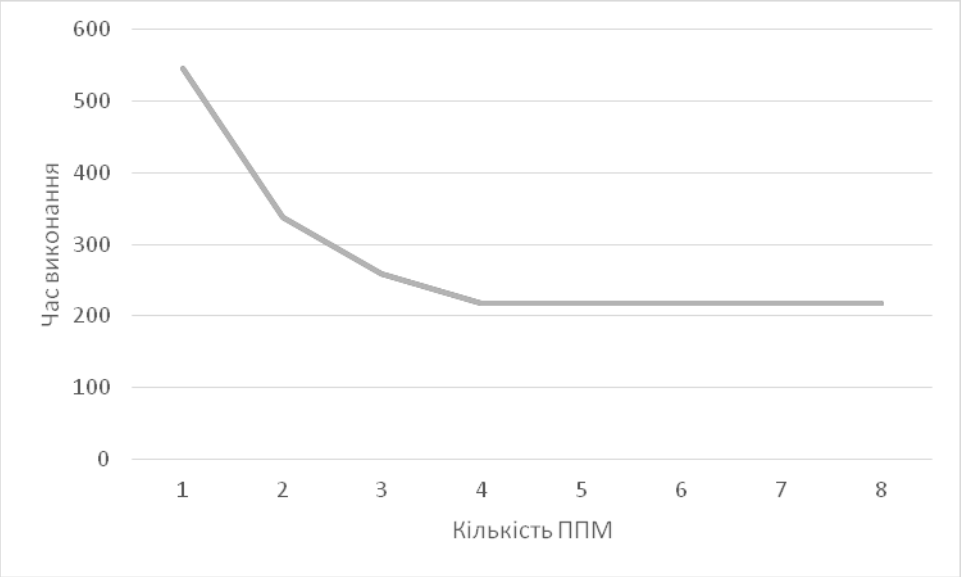


Рис. 25. Часова діаграма обчислення виразу з матриць

На основі графу видно, що є 3 потоки, тобто найшвидше обчислення буде при використанні трьох процесорів. Але оскільки операція транспонування виконується відносно інших швидко, вона після виконання активізує 2 інші вершини. Тобто в системі виконується вже 4 підпрограми паралельно рис. 26, що дасть виграш у часі. Це явище

називається прихованим паралелізмом, яке при статичному розпаралелюванні важко виявити

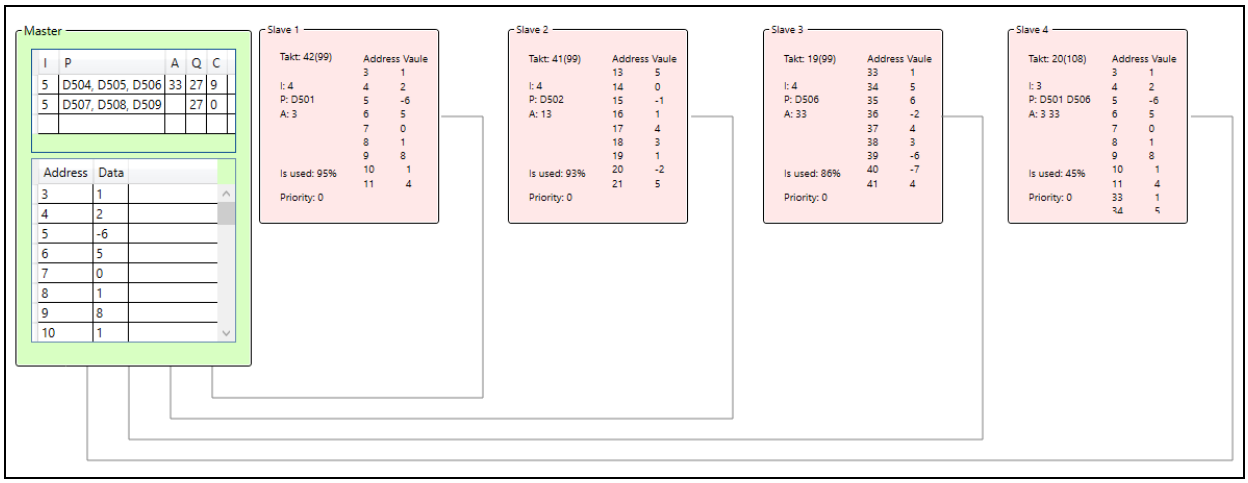


Рис. 26. Виконання 4-х підпрограм паралельно

4.3. Паралельне обчислення двох незалежних крупнозернистих графів

Архітектура системи дозволяє виконування обчислення двох незалежних крупнозернистих графів. Для дослідження показників ефективності було паралельно обчислено крупнозернистий граф розв’язання СЛАР та крупнозернистий граф обчислення алгебраїчного виразу із матрицями.

В таблиці 12 наведені результати обчислення двох графів. Моделювання обчислень відбувалося при різних кількостях підлеглих процесорних модулів, від 1-го до 8-ми процесорів.

Таблиця 12. Результати моделювання розв'язку двох графів

Кількість ППМ	1	2	3	4	5	6	7	8
Час виконання (тактів)	1946	1043	794	751	700	640	602	598

Для порівняння ефективності в таблиці 13 наведені дані моделювання обчислення двох графів паралельно та моделювання обчислення двох графів окремо, в таблиці наводиться сума часу виконання обчислення графів незалежно.

Таблиця 13. Результати моделювання розв'язку двох графів паралельно та окремо

Кількість ППМ	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Обчислення графів паралельно	1946	1043	794	751	700	640	602	598
Обчислення графів окремо	1948	1294	1043	978	975	917	917	917

На рис. 27 наведено графіки отриманих результатів із таблиці 12.

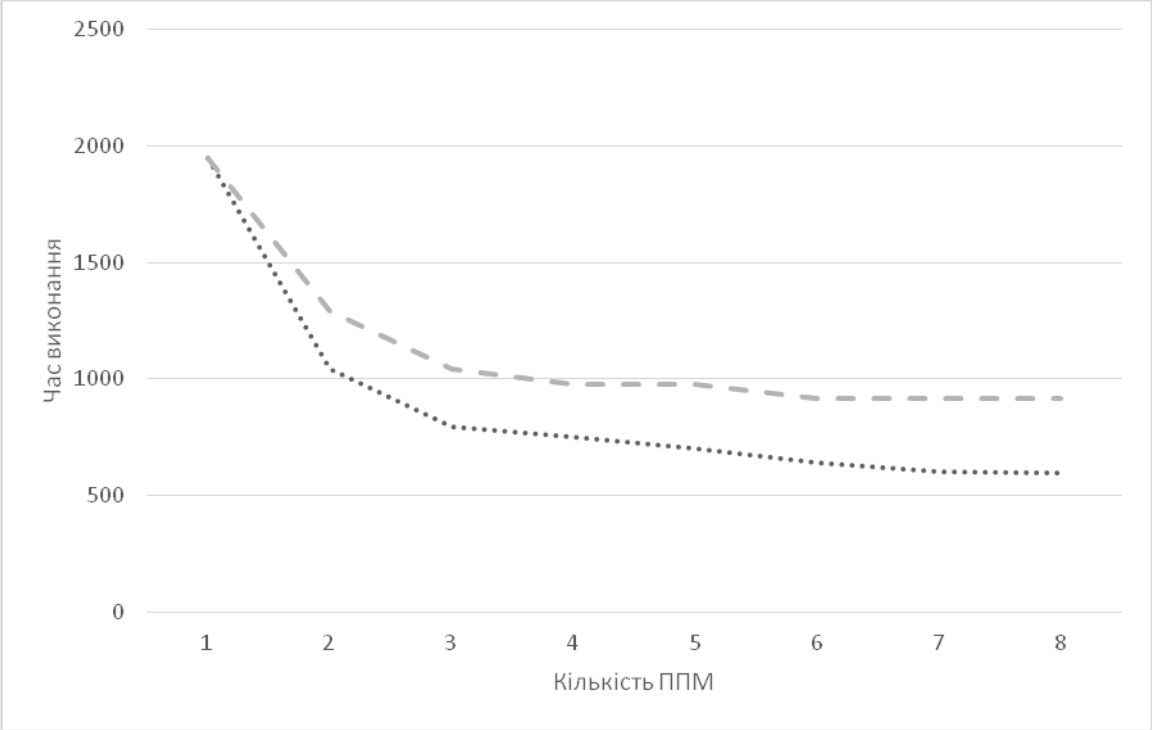


Рис. 27. Результати моделювань обчислень двох графів

В таблиці 14 наведено результати, що показують, який відсоток від загального часу обчислення двох графів конкретний обчислювальний процесор був завантажений.

Таблиця 14. Результати завантаженості процесорів при обчисленні двох графів

Кількість ППМ									Середня завантаженість ППМ
1	98%								98%
2	92%	90%							91%
3	83%	75%	81%						79%
4	69%	62%	59%	62%					63%
5	53%	56%	53%	53%	56%				54%
6	54%	46%	44%	47%	57%	47%			49%
7	48%	45%	40%	49%	38%	56%	38%		44%
8	39%	39%	59%	40%	42%	43%	26%	28%	39%
Номер ППМ	1	2	3	4	5	6	7	8	

З отриманих результатів можна отримати висновок, що запропонована архітектура мультипроцесорної системи та запропонований алгоритм обчислення крупнозернистих графів, що система може ефективно використовуючи свої ресурси розв’язувати дві незалежні задачі.

Для конкретних крупнозернистих графів найшвидшою мультипроцесорною системою є система з 8 обчислювальними процесорами. Слід зазначити, що при 7 процесорах було отримано трохи гірший час, але при цьому процесори використовувалися значно ефективніше. Тому можна зробити висновок, що найефективнішою мультипроцесорною системою є система з 7 підлеглими процесорними модулями.

4.4. Висновки до розділу 4

Були проведені моделювання обчислень крупнозернистих графів в мультипроцесорній системі.

Найефективніша мультипроцесорна система є система з кількістю підлеглими процесорами, яка дорівнює кількості потоків в ярусно-паралельному графі, і збільшення кількості обчислювальних процесорів не покращують результат.

В деяких випадках найефективнішою мультипроцесорною системою може буди система з меншою кількість обчислювальних процесорів, ніж кількість потоків в ярусно-паралельному графі. Такі результати отримуються, якщо в крупнозернистому графі є прихований паралелізм.

Зазвичай час виконання обчислень графа в найефективнішій мультипроцесорній системі буде дорівнювати часу обчислення головної гілки графа.

Для рівномірного розподілу завантаженості обчислювальних процесорів доцільно використовувати пріоритетне призначення готового завдань процесорам.

Пропонована архітектура мультипроцесорної системи дозволяє виконувати обчислення двох незалежних крупнозернистих графів разом, тобто паралельно. Даний підхід дозволив на третину зменшити час обчислення в порівнянні з окремим виконанням двох крупнозернистих алгоритмів.

5. ПОБУДОВА БІЗНЕС МОДЕЛІ

5.1. Опис проблеми

В кінці 20 століття з розвитком технології Інтернет почалося швидкий розвиток біржової торгівля [39]. Майже всі фінансові операції, які раніше здійснювали учасниками ринку особисто в деякому конкретному місці, сьогодні здійснюються через мережу Інтернет, що в свою значно збільшило кількість учасників ринку.

Організація електронної торгівлі безпосередньо стосується обмеженого кола осіб. Проте опосередковано вона має вплив на велику кількість людей. Оскільки в електронних торгах беруть участь великі компанії, державні установи, банки та велика кількість приватних інвесторів. Збої роботи таких систем можуть викликати великі фінансові та репутаційні втрати наведених вище організацій та осіб. Саме тому дані системи потребують великого ступеню надійності програмного забезпечення, що його реалізує. Під надійністю мається на увазі безперервність та відмовостійкість. Також великого значення мають вимоги до швидкодії обробки та передачі даних. Отже програмне забезпечення, яке реалізує роботу з фінансовими активами, має відповідати високим стандартам.

На сьогоднішній день у сфері електронної торгівлі існує низка проблем.

По-перше, щодня зростає кількість учасників електронних торгів, і системам стає все складніше підтримувати належний рівень обслуговування клієнтів. Також зі збільшенням кількості учасників, розширюється географія цих учасників. Адже завдяки електронній торгівлю людина може здійснювати угоди не виходячи с дому, що у свою чергу накладає додаткових вимог до систем електронної торгівлі [40, 41].

По-друге, для організації роботи сучасних електронних ринків, необхідно оброблювати великі обсяги даних, які збільшуються зі зростанням кількості клієнтів та фінансових активів, якими можливо торгувати на ринку. Окрім самої обробки даних, необхідно забезпечувати належний рівень обробки вхідного потоку інформації, інтенсивність якого весь час збільшується.

Для систем підтримки електронних торгів є важливим питання швидкодії. Як будь-яка система реального часу, система електронних торгів має підтримувати належний час реагування на зовнішні події, який сьогодні є досить малим. Окрім цього необхідно забезпечувати однаковий час обробки запитів клієнті для рівноправного обслуговування.

Дуже важливим є забезпечення надійності торгової платформи, з чого випливають високі до відмовостійкості системи на всіх рівнях. Загрози надійності мають високий ризик, через великі збитки, тому необхідно зменшувати ймовірність виникнення цих загроз.

Системи, які працюють з фінансовими активами, повинні відповідати сучасним стандартам забезпечення безпеки. Вони повинні бути захищанні від хакерських атак та шкідливого ПЗ. В свою чергу забезпечення необхідного рівня безпеки впливання на загальну ефективність системи, тому вирішення цієї проблеми не є легкою задачею.

Також розробка та підтримка торгової платформи потребує значних коштів.

Все вище описане можна узагальнити у схемі «Дерево проблем», яка зображена на рис. 28.

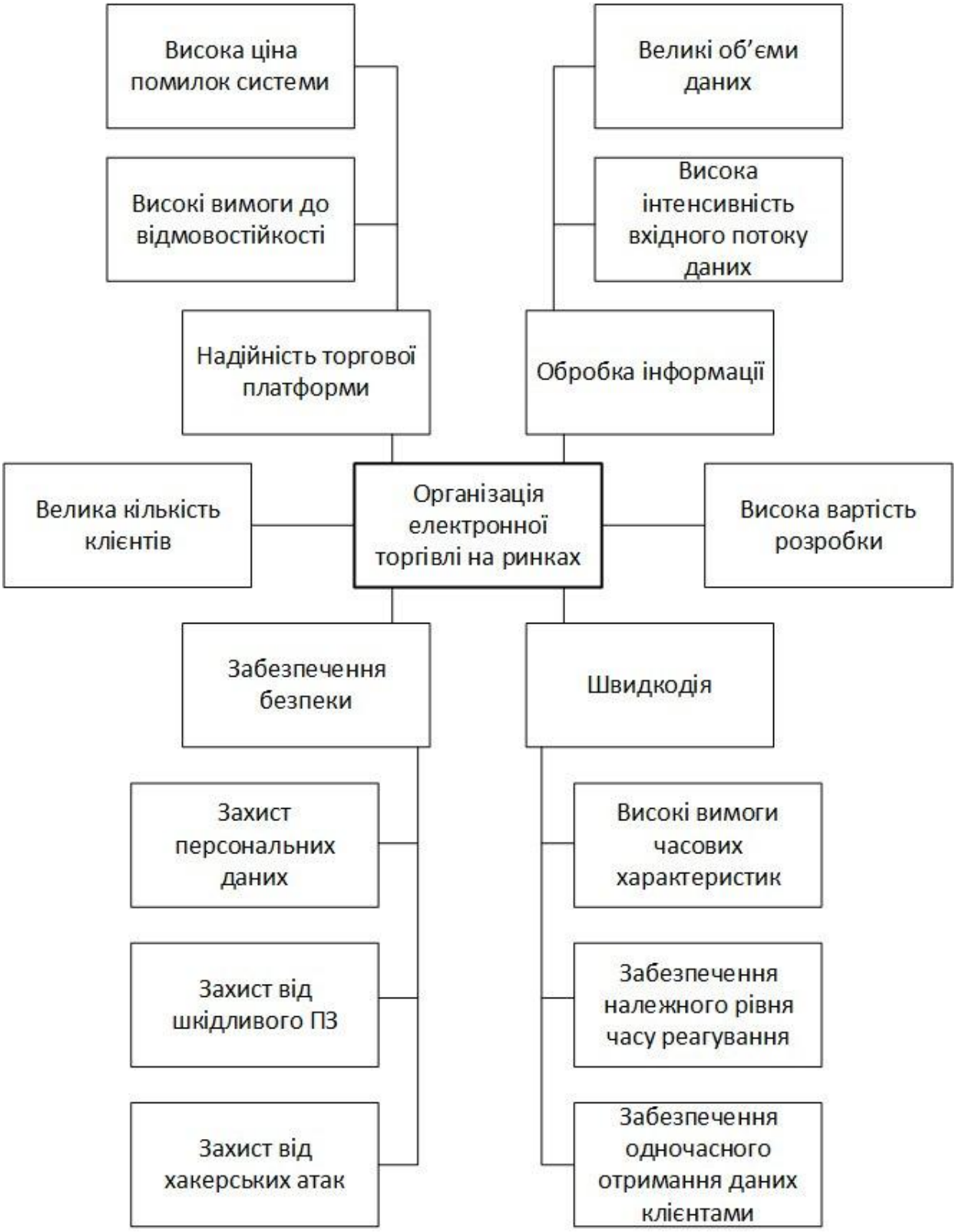


Рис. 28. Дерево проблем

5.2. Зацікавлені сторони

У вирішенні описаної вище проблеми існує декілька зацікавлених сторін.

Найбільш очевидними та впливовими зацікавленими сторонами, є самі торгові біржі та брокери.

Сучасні електронні біржі повинні обслуговувати велику кількість учасників ринку, при цьому необхідно забезпечувати високу якість. Клієнтам потрібно надавати повну інформацію про стан ринку, яка повинна бути зрозуміла навіть початківцям. Власники бірж зацікавлення в збільшені об’ємів та видів фінансових активів, а для цього необхідні відповідні обчислюванні можливості. Також біржі витрачають значні ресурси для забезпечення надійності та безпеки, адже помилки та загрози можуть нанести значних збитків.

Брокери як посередники між приватними інвесторами та торговими біржами також зацікавленні в ефективній платформі організації ринку. Вимоги брокера до цієї платформи менші ніж у бірж, але при цьому її ціна має бути значно менше. Також брокер зацікавлений в ефективній організації обміну інформації між ним та біржою, вона має бути швидкою та надійною. Адже брокери можуть оперувати значними за обсягами фінансовими активами.

Організатори локальних ринків, а саме дилінгові центри, мають потребу в точно таких система як і звичайні біржі. Але як і брокери, потребують системи з нижчим рівнем вимог, тому і меншою ціною.

Банки як одні з найбільших учасників ринку зацікавлення в торгових платформах, які відповідають високому рівню швидкодії, надійності та безпеки, адже вони оперують великими фінансовими активами.

Приватні інвестори зацікавлення в ефективному інструменті торгівлі на біржі на пряму чи через брокера. Він має бути швидким, надійним та інтуїтивно зрозумілим.

Також можна виділити категорію потенційних приватних інвесторів, які зацікавленні в швидкому та зрозумілому отриманні необхідної інформації про стан ринку.

Вище описані зацікавленні сторони можна узагальнити у таблиці 15.

Таблиця 15. Зацікавлені сторони

Зацікавлена сторона	**Інтерес зацікавленої особи**	**Вплив зацікавленої особи**	**Стратегії приваблення зацікавлених сторін**
Власники бірж	Надійний та ефективний спосіб організації біржових торгів. Підвищення швидкості обробки даних. Забезпечення обслуговування для максимальної кількості клієнтів.	Високий	Проведення презентації для представників зацікавлених осіб. Участь у спеціалізованих конференціях та виставках. Забезпечення підтримки системи.
Брокери	Ефективна організація обміну інформації з біржою. Забезпечення захисту інформації.	Високий	
Дилінгові центри	Надійний та ефективний спосіб організації торгів.	Високий	

Продовження таблиці 15. Зацікавлені сторони

Зацікавлена сторона	**Інтерес зацікавленої особи**	**Вплив зацікавленої особи**	**Стратегії приваблення зацікавлених сторін**
Банки	Отримання швидкого, надійного та безпечного інструменту доступу до торгів.	Середній	Проведення презентації для представників зацікавлених осіб. Участь у спеціалізованих конференціях та виставках. Забезпечення підтримки системи.
Приватні інвестори	Отримання швидкого, надійного, зрозумілого інструменту доступу до торгів.	Низький	
Приватні особи	Швидке та зрозуміле отримання необхідної інформації про стан ринку.	Низький	

5.3. Комерційне рішення. Основні характеристики

Для вирішення вище зазначених проблем пропонується розробити продукт, що буде системою реального часу, яка реалізована на потоковій архітектурі. Дана система реалізує досліджувану модель реалізації крупнозернистого паралелізму в мультипроцесорних системах. Дана модель дозволить збільшити швидкість обробки даних та надійність, за рахунок побудови системи відмовостійкості на рівні обробки операцій.

По-перше, збільшиться кількість інформації, яку зможе обробити торгова платформа, збільшиться швидкість обробки. Що дозволить збільшити кількість клієнтів та обсяги фінансових активів. По-друге збільшиться надійність платформи, за рахунок чого біржа, брокер чи дилінговий центр стане більш привабливим для нових клієнтів.

Саме тому, очевидно, що клієнти даного продукту є біржі, брокери та дилінгові центри, а співпраця буде побудована на моделі співробітництва бізнесу для бізнесу, тобто B2B.

Даний продукт буде має в собі програмну та апаратну частину. Апаратна частина буде розроблена на основі потокової системи. Дана розробка є досить складною та потребує нестандартних рішень, що у свою чергу є дорогим, але дозволить збільшити ефективність готового продукту за рахунок врахування особливостей роботи електронної біржі. Також розробка нової архітектури призведе до складнощів реалізації програмної частини, але це також дозволить розробити систему, яка буде ефективно вирішувати специфічні задачі.

5.4. Конкурентні переваги рішення

Основними конкурентами дано продукту є торгові платформи, які реалізовані як системи реального часу на класичній фон-нейманівській архітектурі. Існує велика кількість процесорів та обчислювальних вузлів на основі фон-нейманівській архітектурі, які дозволяю швидко та надійно оброблювати інформацію. Але можливості цієї архітектури обмежені, тому швидкість та надійність постійно зменшуються зі збільшенням об’ємів оброблювальної інформації, і забезпечення необхідної ефективності потребує все більшої кількості ресурсів.

Основні конкуренти мають безліч недоліків, які були описані в пункті 5.1, Майже всі недоліки, що має аналог, вирішуються за допомогою запропонованій моделі реалізації крупнозернистого паралелізму в мультипроцесорних системах.

Реалізація торгової платформи як системи реального часу на потоковій системі дозволить отримати наступні конкурентні переваги:

- збільшення кількості учасників електронного ринку, яких зможе обслуговувати біржа, брокер чи дилінговий центр в порівнянні з існуючими аналогами;

- збільшення швидкодії обробки запитів учасників електронної біржі;
- збільшення надійності торгової платформи на рівні обробки інформації;
- збільшення об’єму обробки інформації.

5.5. Клієнти. Сегменти ринку споживання

Клієнти даного продукту є біржі, брокери та дилінгові центри.

Сегментацію ринку можна провести за обсягом користувачів, яких буде обслуговувати торгова платформа.

Можна виділити великі електронні біржі, які не надають прямого доступу приватним інвесторам, а дозволяють виходити на ринок лише через брокерські фірми. Такі біржі не мають забезпечувати обслуговування великої кількості клієнтів, але при цьому вони оперують великими об’ємами даних та мають підвищенні вимоги до швидкості обробки вхідного та вихідного потоку даних.

Біржі, які надають прямий доступ до торгів приватним інвесторам, обслуговують значно більшу кількість прямих користувачів, але при цьому мать менший об’єм даних. що оброблюється.

Брокерські компанії мають одночасно співпрацювати з торговою біржою та приватними інвесторами. Їх торгові платформи оперують меншими об’ємами даних та обслуговують меншу кількість клієнтів, ніж біржа. Але повинні забезпечувати високу швидкодію обміну інформації як посередник між приватним інвестором та біржою.

Дилінгові центри це аналог невеликої біржі. Вони мають схожі потреби, але на меншому рівні, при цьому потребують більш дешевого продукту.

Окрім сегментації ринку за кількістю клієнтів, можна розділити за типами фінансових активів, якими торгує біржа.

Біржі можуть мати різні інструменти, такі як фондовий ринок, це торгівля акціями та облігаціями, та строковий ринок, торгівля ф’ючерсами

та опціонами. В залежності від цього біржі потребують відповідних торгових платформ [42].

Серед дилінгових центрів можна виділити форекс, міжбанківський міжнародний валютний ринок [43]. Він має свої особливості, тому їх врахування дозволить розробити більш ефективну системи саме для валютних операцій.

5.6. Унікальна ціннісна пропозиція

Ціннісна пропозиція – це пояснення того, як продукт вирішує проблему. У попередніх пунктах вже було детально описано проблеми та шляхи їх вирішення для різних категорій зацікавлених користувачів шляхом використання моделі реалізації крупнозернистого паралелізму в мультипроцесорних системах. Придбання програмного забезпечення дозволить розширити можливості інфраструктури торгової платформи. При цьому ціна такого програмного забезпечення значно менша, ніж ціна придбання нового обладнання для забезпечення відповідних можливостей системи. Окрім ціни обладнання, виникають проблеми у розширенні та підтримки великої інфраструктури. Тому узагальнюючи можна виділити наступне: розробка торгової платформи, як системи реального час на потоковій архітектурі, дозволить збільшити кількість учасників торгів, збільшить швидкодію обробки запитів та збільшити надійність системи в порівнянні з існуючими рішеннями.

5.7. Доходи і витрати

Основним доходом буде дохід за виконання замовлень від бірж, брокерів та дилінгових центрів. Виконання замовлення включає в себе повний цикл розробки апаратної та програмної частини торгових терміналів. Також буде здійснюватися технічна та гарантійна підтримка.

До основних витрат відноситься витрати на закутку необхідних деталей для розробки апаратної частини торгових терміналів. Вартість деталей для однієї торгової платформи може змінюватися в залежності від вимог до платформи.

До загальних витрат відносяться витрати на оренду приміщення, комунальних послуг та витрат на засоби зв’язку.

Ще однією з суттєвих статей витрат є витрати пов’язані з рекламою та маркетингом

Витрати на виплату заробітної плати найманому персоналу без податків складає 50000$ на місяць. З урахуванням податків ця сума буде складати 72000$ на місяць. що в рік буде складати 864000$

Детальніше з планом витрат можна ознайомитися з таблиць 16.

Таблиця 16. Загальні витрати

Список витрат	**Сума в рік , тис. $**	**Періодичність**
Оренда	60	Рік
Реклама та маркетинг	36	Рік
Комунальні послуги	6,4	Рік
Оплата послуг інтернет-провайдерів	3,6	Рік
Результат:	**106**	Рік

Прибутки очікуються на другому році від продажу продукту. Детальніше ознайомитися зі зведеним планом прибутків та витрат можна з таблиці 17.

Таблиця 17. Фінансовий результат діяльності

Найменування витрат	**1-й рік, т. $**	**2-й рік, т. $**	**3-й рік, т. $**	**4-й рік, т. $**	**5-й рік, т. $**	**Загальні результати, т. $**
Загальні витрати	102	106	106	106	106	526
Замовлення деталей	300	400	550	600	600	2450
ЗП	792	864	864	864	864	4248
Витрати	**1194**	**1370**	**1520**	**1570**	**1570**	**7224**
Заплановані прибутки	**760**	**1400**	**1850**	**2300**	**2400**	**8710**
Результат(без оподаткування)	**-434**	**30**	**330**	**730**	**830**	**1486**

5.8. Бізнес модель

Узагальнимо, все написане вище у лаконічну бізнес-модель у вигляді lean canvas.

Таблиця 18. Бізнес-модель lean canvas

<table>
<tr>
<td rowspan="2">Проблема
Обмеженість у кількості клієнтів, яких можна обслуговувати
Не достатня швидкодія
Складність забезпечення надійності та безпеки</td>
<td>Рішення
Розробка торгової платформ, як системи реального час на потокової архітектури</td>
<td rowspan="2">Унікальна ціннісна пропозиція
Розширення можливостей інфраструктури торгової платформи за рахунок встановлення унікально ПЗ, а не придбання нової апаратури.</td>
<td>Прихована перевага
Забезпечення надійності роботи на рівні обробки операції</td>
<td rowspan="2">Споживачі
Біржі, брокери, дилінгові центри</td>
</tr>
<tr>
<td>Ключові метрики
Кількість проданих одиниць продукту кожного виду</td>
<td>Канали
Керівники відповідних організації</td>
</tr>
<tr>
<td colspan="3">Структура витрат
Закупка деталей для системи
Утримання персоналу
Утримання офісу
Податкові витрати</td>
<td colspan="2">Потоки доходів
Доходи від продажу та підтримки продукту</td>
</tr>
</table>

Отже, зважаючи на написане вище, можна зробити висновок, що даний продукт можна реалізувати та на основі цього побудувати в подальшому бізнес. Наведені розрахунки не є точними, адже не враховують всіх ризиків та специфіки оподаткування, проте побудована бізнес-модель вказує на життєздатність проекту.

5.9. Висновки до розділу 5

У даному розділі було виділено основні проблеми, що існують у електронної торгівлі на ринках, та підсумовано їх у дереві проблем. Також

були розглянуті зацікавлені сторони у вирішенні даних проблем та визначення їх впливу на подальше вирішення даних проблем, проведено дослідження потенційних клієнтів та сегменту ринку споживання. У якості комерційного рішення було запропоновано продукт, який базується на запропонованій у даній дисертації моделі реалізації крупнозернистого паралелізму в мультипроцесорних системах, що вирішує наявні проблеми та враховує інтерес зацікавлених осіб. Було визначено конкурентні переваги та унікальну ціннісну пропозицію запропонованого продукту. На основі наведених вище досліджень було спрогнозовано потенційні доходи і витрати. Як результат була сформована бізнес-модель, що доводить потенціал даного продукту.

ВИСНОВКИ

Метою даної магістерської дисертації було дослідження можливості прискорення обробки інформації за рахунок динамічного розподілу робіт між обчислювальними вузлами в мультипроцесорних системах.

Аналіз існуючих рішень показав, що на даний момент універсальні системи моделювання мультипроцесорних систем є не доступними для невеликих організацій, компаній та підприємств. Саме тому дослідження, що проведено в даній роботі є актуальним для спеціалізованих вузько направлених областей.

Була досліджена можливість підвищення ефективності обробки даних в мультипроцесорних системах під керівництвом потоків даних за рахунок динамічного розподілення робіт між процесорами. Запропонована модель реалізації системи, що дозволяє практично вдвічі зменшити кількість звернень до загального ресурсу при пересиланні даних, що дає потенційну можливість прискорити обробку інформації. Була запропонована архітектура мультипроцесорної системи для обчислення крупнозернистих графів, час обчислення в якій був зменшений за рахунок зменшення часу на пересилку даних для виконання підпрограм.

Зокрема було встановлено, що для дослідження параметрів мультипроцесорної системи необхідно створити настільне застосування імітаційної моделі. При аналізі засобів розроблення настільних застосувань були обрано: мова програмування C#, WPF .NET Framework, середовище розробки Microsoft Visual Studio 2012.

Розроблена імітаційна модель крупнозернистого паралелізму в мультипроцесорних системах дозволила дослідити:

- параметри найефективнішої мультипроцесорної системи для обчислення крупнозернистого алгоритму;

- вплив прихованого паралелізму на швидкість розв’язання алгоритмів;
- досягнення рівномірної завантаженості між обчислювальними процесорами;
- процес обчислення та параметри мультипроцесорної системи при одночасному розв’язанні декількох незалежних крупнозернистих алгоритмів. Паралельне виконання двох незалежних крупнозернистих алгоритмів дозволило на третину зменшити час обчислення в порівнянні з окремим виконанням алгоритмів.

Було запропоновано бізнес модель впровадження розробленої моделі мультипроцесорної системи. Було виділено основні проблеми, що існують у електронної торгівлі на ринках, та підсумовано їх у дереві проблем. Також були розглянуті зацікавлені сторони, потенційні клієнти та сегменти ринку споживання. У якості комерційного рішення було запропоновано продукт. Було визначено конкурентні переваги та унікальну ціннісну пропозицію запропонованого продукту.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. «Dataflow-архитектуры». [Електронний ресурс]. — Режим доступу: https://habrahabr.ru/post/122479/ — Дата доступу : Листопад 2017. — Назва з екрана.
2. Таненбаум Э. Современные операционные системы [Текст] / Таненбаум Э., Бос Х : пер. с англ — СПб.: БХВ-Петербург, 2015. – 123 с.
3. Воеводин В.В. Параллельные вычисления [Текст] / В.В.Воеводин, Вл.В.Воеводин. — СПб.: БХВ-Петербург, 2002. – 608 с.
4. Соколов А. Методы и алгоритмы параллельных вычислений [Текст] / Соколов А.В., Барковский Е.А., Кучумов Р.И., Сазонов А.М — Петрозаводск: ПетрГУ, 2016. – 48 с.
5. Шпаковский Г.И. Программирование для многопроцессорных систем в стандарте MPI [Текст] / Г.И.Шпаковский, Н.В.Серикова. — Мн.: БГУ, 2002. — 323 с.
6. Барский А.Б. Архитектура параллельных вычислительных систем 2-е изд. [Текст] / Барский А.Б. — М.: НОУ "Интуит", 2016. — С. 98-112.
7. Homayoun N. Dynamic priority scheduling of periodic and aperiodic tasks in hard real-time systems [Text] / N.Homayoun, P.Ramanathan — Boston : Kluwer Academic Publishers, 1994. — P.207-232.
8. Elsadek A.A. Heuristic model for task allocation in heterogeneous distributed system [Text] / A.A.Elsadek, B.E.Wells — 1996. — P. 659-671.
9. Пиза Н.Д. Исследование эффективности применения параллельных вычислительных систем для моделирования движения космического аппарата [Текст] / Н.Д.Пиза — СПб.: Питер, 2003.— С. 102-108.
10. Фритч В. Применение микропроцессоров в системах управления [Текст] / В.Фритч. — М.: Мир, 1984. — 464 с.

11. Тарасик В.П. Математическое моделирование технических систем [Текст] / В.П.Тарасик. — Мн.: Дизайн ПРО, 2004. — 640 с.
12. Журавлев Ю.П. Системное проектирование управляющих ЦВМ [Текст] / Ю.П.Журавлев. — М: Сов. радио, 1974. — 368 с.
13. Молчанов И.Н. Введение в алгоритмы параллельных вычислений [Текст] / И.Н.Молчанов. — К.: Наукова думка, 1990. — 128 с.
14. Сосонкин В.Л. Микропроцессорные системы числового прог¬раммного управления [Текст] / В.Л.Сосонкин.— М.: Машиностроение, 1985.— 288 с.
15. Жабина, В.В. Методы и средства повышения эффективности реализации мелкозернистого параллелизма в системах реального времени [Текст] : дис. канд. техн. наук : захищ. 20.06.11 / Валентина Валериевна Жабина. — К., 2011. — С. 22-30.
16. «Xilinx». [Електронний ресурс]. — Режим доступу: https://uk.wikipedia.org/wiki/Xilinx — Дата доступу : Листопад 2017. — Назва з екрана.
17. «Altera». [Електронний ресурс]. — Режим доступу: https://ru.wikipedia.org/wiki/Altera — Дата доступу : Листопад 2017. — Назва з екрана.
18. «Altera». [Електронний ресурс]. — Режим доступу: https://www.altera.com/ — Дата доступу : Листопад 2017. — Назва з екрана.
19. Dennis J. B. A preliminary architecture for basic data flow processor [Text] / J.B.Dennis, D.P.Missunas — N.Y.: IEEE, 1975. — P. 126-132.
20. Silva J.G.D. Design of processing subsystems for Manchester data flow computer [Text] / J.G.D.Silva, J.V.Wood — N.Y.: IEEE, 1981. — P. 218-224.
21. Майерс Г. Архитектура современных ЭВМ [Текст] / Г.Майерс: пер с англ. — М.: Мир, 1985. — 312 с.

22. Жабин В.И. Метод распараллеливания процессов в вычислительных системах [Текст] / В.И.Жабин // Вісник Національного технічного університету України “Київський політехнічний інститут”. Інформатика, управління та обчислювальна техніка. — 2000. — № 34. — С. 136-142.
23. Жабин В.И. Реализация параллельных процессов в вычислительных системах [Текст] / В.И.Жабин // Искусственный интеллект. — 2002. — №3. — С. 235-241.
24. Жабин В.И. Реализация вычислений под управлением дескрипторов данных в мультипроцессорных системах [Текст] / В.И.Жабин // Электронное моделирование. — 2003. — Т. 25, № 1. — С. 35-47.
25. Жабин В.И. Архитектура вычислительных систем реального времени [Текст] / В.И.Жабин. — К.: ТОО “ВЕК+”, 2003. — 176 с.
26. Максфилд К. Проектирование на ПЛИС. Архитектура, средства и методы [Текст] / К.Максфилд. — М.: Изд. дом «Додэка-XXI», 2007 — 408 с.
27. Жабин В.И. Графическое описание архитектуры вычислительных систем [Текст] / В.И.Жабин // Вісник Національного технічного університету України “КПІ”. Інформатика, управління та обчислювальна техніка. — К: “ВЕК+”, 2001. — № 36. — С. 80–88.
28. Гуров, В.О. Прискорення динамічного розподілу завдань в мультипроцесорних система [Текст] / В.О. Гуров, В.В. Жабіна // VIII Міжнародній конференції студентів і молодих учених "Сучасні Інформаційні Технології 2018" — Одеса, 2018 — С. 1-5.
29. Гуров, В.О. Модель реалізації крупнозернистого паралелізму в мультипроцесорних система [Текст] / В.О. Гуров, В.В. Жабіна // Наукова конференція магістрантів та аспірантів «Прикладна математика та комп’ютинг» ПМК-2018 — Київ, 2018. — С. 1-5.

30. Mike Ebbers Introduction to the New Mainframe: z/OS Basics [Text] / Mike Ebbers, John Kettner, Wayne O'Brien, Bill Ogden.— 3rd edition. — IBM Redbooks, 2012. — 96 p.
31. Буч, Г. Язык UML. Руководство пользователя. 2-е изд. [Текст] / Г. Буч, Д. Рамбо, И. Якобсон : пер. з англ. — М. : ДМК Пресс, 2006. — С. 47-49.
32. Вигерс, К.И. Разработка требований к программному обеспечению [Текст] / К.И. Вигерс : пер. з англ. — М. : Издательско-торговый дом «Русская Редакция», 2004. — С. 7-9.
33. Вимоги до ПЗ [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/Вимоги_до_програмного_забезпечення. — Дата доступу : Вересень 2017. — Назва з екрана.
34. Пример написания функциональных требований к Enterprise-системе [Електронний ресурс]. — Режим доступу : https://habrahabr.ru/post/245625. — Дата доступу : Вересень 2017. — Назва з екрана.
35. Діаграма прецедентів [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/Діаграма_прецедентів. — Дата доступу : Вересень 2017. — Назва з екрана.
36. UML-схемы последовательностей: справочные материалы [Електронний ресурс]. — Режим доступу : https://msdn.microsoft.com/uk-ua/library/dd409377.aspx. — Дата доступу : Вересень 2017. — Назва з екрана.
37. Діаграма послідовності [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/Діаграма_послідовності. — Дата доступу : Вересень 2017. — Назва з екрана.
38. Теория и практика UML. Диаграмма последовательности [Електронний ресурс]. — Режим доступу : http://it-gost.ru/articles/view_articles/94. — Дата доступу : Вересень 2017. — Назва з екрана.

39. «Развитие системы электронной биржевой торговли» [Електронний ресурс]. — Режим доступу: http://www.kandinskaya.narod.ru/kniga2_7_r.html — Дата доступу : Листопад 2017. — Назва з екрана.
40. «2016 Stock Market Investor Profile» [Електронний ресурс]. — Режим доступу: http://www.pseacademy.com.ph/LM/investors~details/id-1498096241729/2016_Stock_Market_Investor_Profile.html — Дата доступу : Листопад 2017. — Назва з екрана.
41. «Статистика по клиентам» [Електронний ресурс]. — Режим доступу: http://www.moex.com/s719 — Дата доступу : Листопад 2017. — Назва з екрана.
42. «Українська біржа». [Електронний ресурс]. — Режим доступу: https://uk.wikipedia.org/wiki/ Українська_біржа — Дата доступу : Листопад 2017. — Назва з екрана.
43. «Форекс» [Електронний ресурс]. — Режим доступу: https://uk.wikipedia.org/wiki/форекс — Дата доступу : Листопад 2017. — Назва з екрана.

ДОДАТКИ

Додаток 1

Копія презентації

Модель реалізації крупнозернистого паралелізму в мультипроцесорних системах

СТУДЕНТ: ГУРОВ В.О.
НАУКОВИЙ КЕРІВНИК: К.Т.Н. ЖАБІНА В.В.

Мета

- Дослідження можливості прискорення обробки інформації за рахунок динамічного розподілу робіт між обчислювальними вузлами в мультипроцесорних системах. Результати дослідження дадуть змогу розробити програмний емулятор для вивчення характеристик в різних режимах реалізації паралельних крупнозернистих алгоритмів.

Об'єкт:

Процеси обробки інформації в паралельних системах реального часу, що пов'язані з проблемами підвищення швидкодії.

Предмет:

Методи і засоби прискорення обчислень в паралельних системах за рахунок динамічного розпаралелювання процесів на рівні обробки завдань та засоби їх моделювання.

СТАТИЧНІ ЗАСОБИ РОЗПАРАЛЕЛЮВАННЯ

Мови програмування (C#, Java, ADA)

Бібліотеки (OpenMP, PVM, MPI)

Недоліки:

- Складність підготовки програм,
- На етапі розробки програм не завжди вдається виявити паралельні гілки, а саме прихований паралелізм,
- Враховується конфігурації апаратних засобів системи

5
(24)

ДИНАМІЧНІ ЗАСОБИ РОЗПАРАЛЕЛЮВАННЯ

- Мови для систем, що керуються потоком даних (mpC, DVM, VAL)

Недоліки:

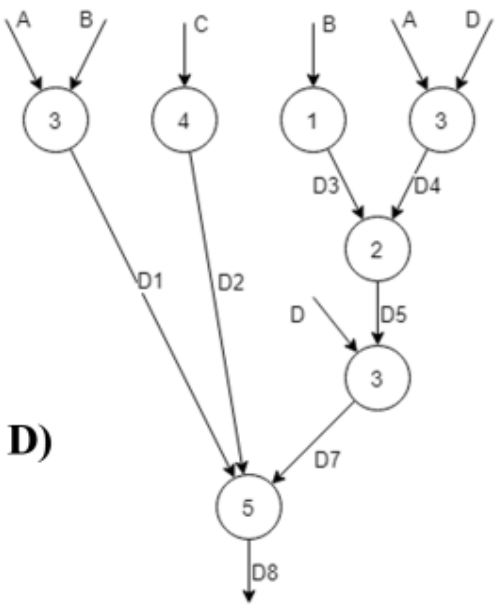
- Не розвинуті на достатньому рівні, навіднінно від статичних засобів
- Спеціалізовані, залежні від системи та задач для яких розробляються

6
(24)

Терміни та скорочення

- Потокова система
- Крупнозернистий паралелізм
- УПМ – управляючий процесорний модуль
- ППМ – підлеглий процесорний модуль

Крупнозернистий граф



$$\mathbf{A} \times \mathbf{B} + \mathbf{C}^2 + \mathbf{D} \times (\mathbf{B}^T + \mathbf{A} \times \mathbf{D})$$

Дескриптор завдання

$$W_i = \{N_i, I_i, P_i, Q_i\},$$

де:

N_i – унікальне для задачі ім'я даного дескриптора;

I_i – ідентифікатор завдання;

P_i – множина імен вихідних даних;

Q_i – сумарне число вхідних потоків даних для i–го завдання.

Дескриптор потоку даних та заявка на виконання завдання

$$D_{ij} = \{N_i, n_{ji}, Q_i, A_{ji}\},$$

де

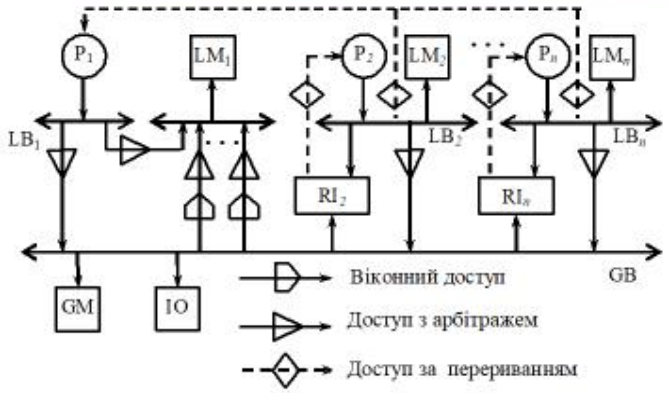
A_{ji} – елемент адресації даних, що визначає місце даних у пам'яті системи.

$$Z_i = \{I_i, P_i, A_i\}$$

де

A_i – множина всіх елементів адресації даних для і-го завдання.

Архітектура системи



11
(24)

Сучасні рішення

- Швидкі та точні
- Енергоефективні
- Універсальні

Недоліки

- Дорогі
- Універсальні

ALTERA
now part of Intel

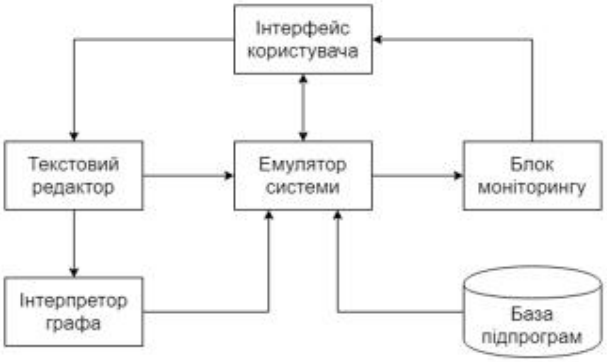
XILINX
ALL PROGRAMMABLE

12
(24)

Каскадна модель

13
(24)

Архітектура ПЗ імітаційної моделі



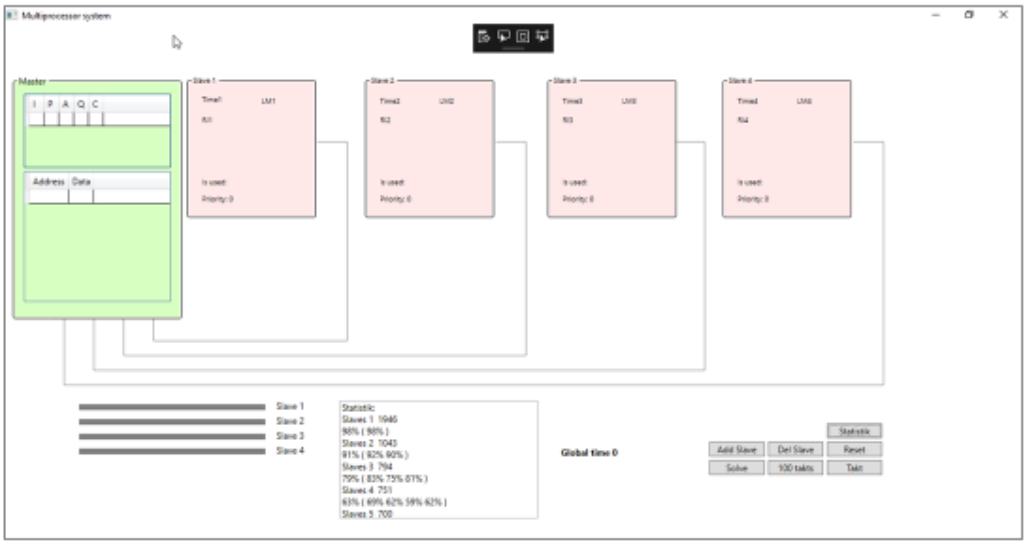
14
(24)

Засоби реалізації

- Windows Presentation Foundation
- C#

Інтерфейс системи

15 (24)

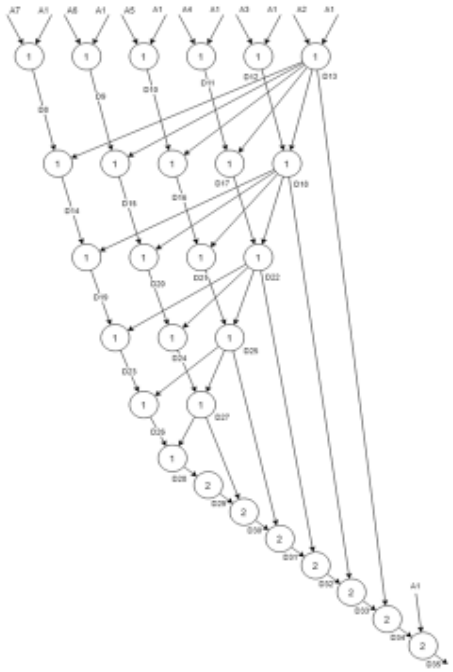


16 (24)

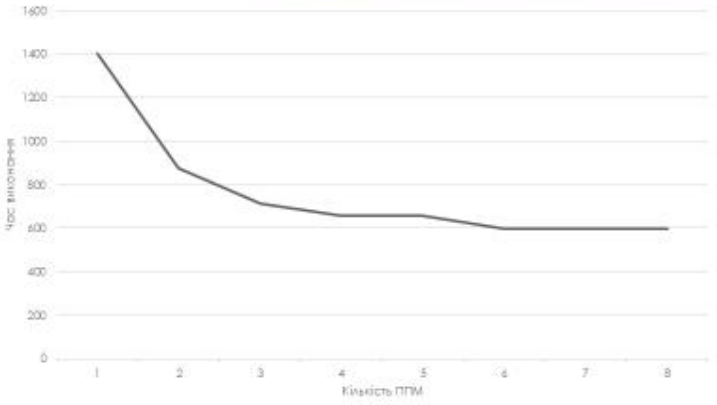
Тестові задачі. Параметри системи

- кількість ППМ (від 1 до 8);
- типи крупнозернистих графів

Граф



Результати моделювання розв'язку СЛАР



Тип системи	Час
Обчислення виконує УПМ	1080
1 УПМ і 1 ППМ	1402
1 УПМ і 2 ППМ	876

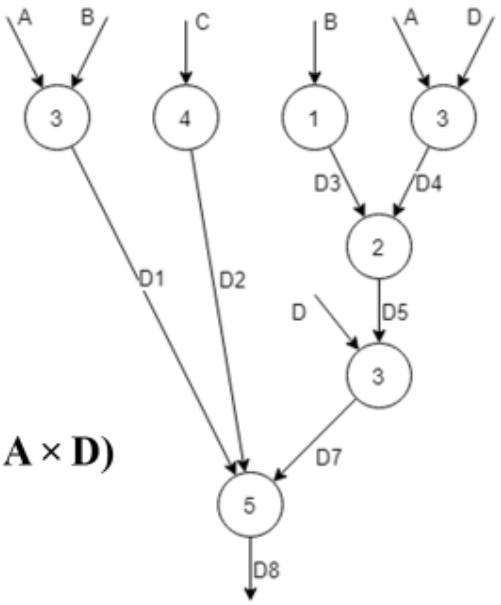
Результати завантаженості ППМ

Кількість ППМ									Середнє значення
1	97%								97%
2	97%	59%							78%
3	96%	50%	44%						63%
4	96%	46%	40%	25%					51%
5	96%	36%	31%	25%	18%				41%
6	95%	40%	34%	28%	20%	10%			37%
7	95%	40%	34%	28%	20%	10%	0%		32%
8	95%	40%	34%	28%	20%	10%	0%	0%	28%
Номер ППМ	1	2	3	4	5	6	7	8	

Кількість ППМ									Середнє значення
1	97%								97%
2	77%	79%							78%
3	61%	63%	67%						63%
4	53%	54%	50%	50%					51%
5	38%	44%	43%	46%	35%				41%
6	40%	42%	43%	41%	30%	30%			37%
7	29%	30%	32%	34%	33%	35%	34%		32%
8	34%	35%	36%	36%	22%	20%	21%	21%	28%
Номер ППМ	1	2	3	4	5	6	7	8	

Граф

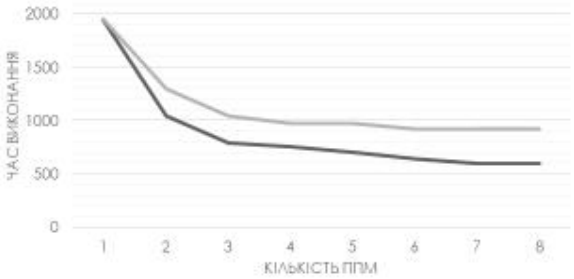
$$\mathbf{A \times B + C^2 + D \times (B^T + A \times D)}$$



21
(24)

Паралельне обчислення двох незалежних крупнозернистих графів

Кількість ППМ	1	2	3	4	5	6	7	8
Обчислення графів паралельно	1946	1043	794	751	700	640	602	598
Обчислення графів окремо	1948	1294	1043	978	975	917	917	917



22
(24)

Наукова новизна

- Одержано подальший розвиток метод динамічного розпаралелювання обчислень у мультипроцесорних системах на рівні програмних модулів, заснований на механізмі формування заявок під управлінням дескрипторів функцій і даних, що у порівнянні з відомими методами дозволяє прискорити реалізацію алгоритмів, в тому числі з неявним паралелізмом.
- Запропоновано програмну модель паралельних обчислень, що на відміну від відомих моделей враховує особливості організації адресного простору кожного процесора та способи синхронізації процесів обміну даними між компонентами системи, що дозволяє на основі моделі визначити можливість підвищення ефективності обчислень при різних вихідних умовах.

Апробація роботи

- VIII Міжнародній конференції студентів і молодих учених "Сучасні Інформаційні Технології 2018" (Одесса, 2018 р.)
- наукова конференція магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2018 (Київ, 2018 р.)

Висновки

- Була досліджена можливість підвищення ефективності обробки даних в мультипроцесорних системах шлях зменшення часу на пересилку даних
- Розроблене ПЗ моделі мультипроцесорної системи, яке дозволило дослідити параметри роботи відповідної системи.

Додаток 2

Лістинги програми

Лістинг 1. Клас, що описує роботу управляючого процесора

```
class MasterProcessor
{
    public List<Job> ListJobs { get; set; }
    public List<Data> ListData { get; set; }

    List<Data> ListAllDataPatern;
    public List<CreateRequestElement> ListRequests { get; set; }
    public Dictionary<int, double?> ListResults { get; set; }
    public List<SlaveProcessor> ListSlaveProcessors { get; set; }

    public int GlobalTime;

    public MasterProcessor() { }

    public bool InsertNewJob(Job newJob)
    {
        ListJobs.Add(newJob);

        int availableDataCount = 0;
        List<int> listAvailableData = new List<int>();
        foreach (int dataIdentifier in newJob.ListInputDataIdentifiers)
        {
            Data availableData = ListData.FirstOrDefault(d => d.Name ==
dataIdentifier && d.DataAddress != -1);
            if (availableData != null)
            {
                availableDataCount += availableData.CountData;
                listAvailableData.Add(availableData.DataAddress);
            }
        }

        ListRequests.Add(
            new CreateRequestElement
                (new Request(newJob.TaskIdentifier,
newJob.ListInputDataIdentifiers, listAvailableData,
newJob.ListOutputDataIdentifiers, newJob.Time),
                newJob.CountInputData,
                availableDataCount));
        return true;
    }

    public bool InsertData(Data newData)
    {
        ListData.Add(newData);
        foreach(CreateRequestElement elem in ListRequests)
        {

if(elem.CreatedRequest.ListInputDataIdentifiers.Contains(newData.Name))
            {
                elem.CreatedRequest.ListDataAddresses.Add(newData.DataAddress);
                elem.CounterInputData += newData.CountData;
            }
        }
        return true;
    }
```

```
    public bool RunTakt()
    {
        GlobalTime++;
        CreateRequestElement activationRequest = null;
        foreach (CreateRequestElement elem in ListRequests)
        {
            if (elem.CounterInputData == elem.CountInputData)
            {
                activationRequest = elem;
                break;
            }
        }

        if(activationRequest!=null)
            ActivationRequest(activationRequest);

        foreach (SlaveProcessor proc in ListSlaveProcessors)
        {
            proc.RunTakt();
            if(proc.FlagSloveTask)
            {
                proc.FlagSloveTask = false;
                ReadDataForSlave(proc);
            }
            if(proc.CurrentTakt == proc.TimeLoadData && proc.FlagRunTask)
            {
                WriteDataToSlave(proc);
            }
        }

        return true;
    }

    public bool ActivationRequest(CreateRequestElement ActivationedRequest)
    {
        SlaveProcessor priority_max = null;
        foreach (SlaveProcessor proc in ListSlaveProcessors)
        {
            if (proc.InputRegistr == null)
            {
                if (priority_max == null || priority_max.priority < proc.priority)
                    priority_max = proc;
            }
        }

        if (priority_max != null)
        {
            priority_max.InputRegistr = ActivationedRequest.CreatedRequest;
            priority_max.TimeLoadData = ActivationedRequest.CountInputData;
            ListRequests.Remove(ActivationedRequest);
        }
        return true;
    }

    public bool ReadDataForSlave(SlaveProcessor proc)
    {
        for (int i = 0; i < proc.ListOutputData.Count; i++)
        {
            Data Paterndata = ListAllDataPatern.FirstOrDefault(d => d.Name ==
proc.ListOutputData.ElementAt(i).Key);
```

```
            Data data = new Data(Paterndata.Name, Paterndata.ArcIndex,
Paterndata.CountData, Paterndata.DataAddress);

            int address = ListResults.Keys.Max();
            ListResults.Add(++address, null);
            data.DataAddress = address + 1;
            foreach (double val in proc.ListOutputData.ElementAt(i).Value)
            {
                ListResults.Add(++address, val);
            }

            InsertData(data);
        }
        return true;
    }

    public bool WriteDataToSlave(SlaveProcessor proc)
    {
        foreach (int id in proc.InputRegistr.ListInputDataIdentifiers)
        {
            for (int i = ListData.First(d => d.Name == id).DataAddress; i <
ListResults.Count + 1; i++)
            {
                if (ListResults[i] != null)
                {
                    proc.ListInputData.Add(i, ListResults[i]);
                    //ListResults[i] = null;
                }
                else
                    break;
            }
        }
        return true;
    }

    public bool AddSlave()
    {
        if (ListSlaveProcessors.Count < 8)
            ListSlaveProcessors.Add(new SlaveProcessor() { GeneralTimeHasTask = 0
});
        return true;
    }

    public bool RemoveSlave()
    {
        if(ListSlaveProcessors.Count > 1)
            ListSlaveProcessors.RemoveAt(ListSlaveProcessors.Count - 1);
        return true;
    }
}
```

Лістинг 2. Клас, що описує роботу пристрою введення та введення

```
class InputOutput
{
    public List<Job> ListJobs;
    public List<Tuple<Data, List<double>>> ListData;
    public InputOutput() { }

    public Job InputJob()
    {
        if (ListJobs.Count > 0)
        {
            Job first = ListJobs.First();
            ListJobs.Remove(first);
            return first;
        }
        return null;
    }

    public Tuple<Data, List<double>> InputData()
    {
        if (ListData.Count > 0)
        {
            Tuple<Data, List<double>> first = ListData.First();
            ListData.Remove(first);
            return first;
        }
        return null;
    }
}
```

Лістинг 3. Клас, що описує роботу підлеглого процесора

```
class SlaveProcessor
{
    public Dictionary<int, double?> ListInputData { get; set; }
    public Dictionary<int, List<double>> ListOutputData { get; set; }
    public Request InputRegistr { get; set; }
    public bool FlagRunTask { get; set; }
    public bool FlagSloveTask { get; set; }

    public int priority { get; set; }
    public int TimeLoadData { get; set; }
    public int TimeRunTask { get; set; }
    public int CurrentTakt { get; set; }
    public int GeneralTimeHasTask { get; set; }

    public bool LoadTask()
    {
        //TimeLoadData = InputRegistr.ListDataAddresses.Count; master set this
value
        ListInputData = new Dictionary<int, double?>();
        TimeRunTask = InputRegistr.Time;
        CurrentTakt = 0;
        priority = 0;
        FlagRunTask = true;
        FlagSloveTask = false;
        return true;
    }

    public bool RunTakt()
    {
        if (FlagRunTask)
        {
            GeneralTimeHasTask++;
            CurrentTakt++;
            if (CurrentTakt == TimeRunTask + TimeLoadData)
                Finalization();
        }
        else if(InputRegistr != null)
        {
            LoadTask();
        }
        else
        {
            priority++;
        }
        return true;
    }

    public bool Finalization()
    {

        ListOutputData = new Dictionary<int, List<double>>();
        foreach (int id in InputRegistr.ListOutputDataIdentifiers)
        {
            List<double> data = new List<double>();
            foreach (double d in ListInputData.Values.Where(d => d != null))
                data.Add(d);
            ListOutputData.Add(id,
```

```
LibraryPrograms.Programs[InputRegistr.TaskIdentifier - 1](data));
        }


        CurrentTakt = 0;
        InputRegistr = null;
        ListInputData = null;
        FlagRunTask = false;
        FlagSloveTask = true;

        return true;
    }
}
```

Лістинг 4. Бібліотека підпрограм

```
static class LibraryPrograms
{
    public delegate List<double> ProgramN(List<double> data);

    static List<double> GaussProgram1(List<double> data)
    {
        List<double> a1 = data.GetRange(0, data.Count / 2);
        List<double> a2 = data.GetRange(data.Count / 2, data.Count / 2);
        double coef = a1[0] / a2[0];
        return a1.Zip(a2, (f, s) => f - s * coef).ToList().GetRange(1, a1.Count -
1);
    }

    static List<double> GaussReturnProgram1(List<double> data)
    {
        List<double> a1 = data.GetRange(0, data.Count / 2);
        List<double> a2 = data.GetRange(data.Count / 2, data.Count / 2);

        double sum = 0;
        for (int i=1; i < a1.Count; i++)
        {
            sum += a1[i] * a2[i];
        }

        a2[0] = (a2[0] - sum) / a1[0];

        return a2;
    }

    static List<double> MatrixProgram1(List<double> data)//trans
    {
        int size = (int)Math.Sqrt(data.Count);
        double[,] matrixA = new double[size, size];
        double[,] matrixB = new double[size, size];
        List<double> result = new List<double>();

        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                matrixA[j, i] = data[i * size + j];
            }
        }

        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                result.Add(matrixA[i, j]);
            }
        }

        return result;
    }

    static List<double> MatrixProgram2(List<double> data)//summ 2
    {
```

```
    int size = (int)Math.Sqrt(data.Count / 2);
    double[,] matrixA = new double[size, size];
    double[,] matrixB = new double[size, size];
    double[,] matrixC = new double[size, size];
    List<double> result = new List<double>();

    for (int i =0; i< size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            matrixA[i, j] = data[i * size + j];
            matrixB[i, j] = data[data.Count / 2 + i * size + j];
        }
    }

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            matrixC[i, j] = matrixA[i, j] + matrixB[i, j];
            result.Add(matrixC[i, j]);
        }
    }

    return result;
}

static List<double> MatrixProgram3(List<double> data)//multipl
{
    int size = (int)Math.Sqrt(data.Count / 2);
    double[,] matrixA = new double[size, size];
    double[,] matrixB = new double[size, size];
    double[,] matrixC = new double[size, size];
    List<double> result = new List<double>();

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            matrixA[i, j] = data[i * size + j];
            matrixB[i, j] = data[data.Count / 2 + i * size + j];
        }
    }

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            for (int k = 0; k < size; k++)
            {
                matrixC[i, j] += matrixA[i, k] * matrixB[k, j];
            }
            result.Add(matrixC[i, j]);
        }
    }

    return result;
}

static List<double> MatrixProgram4(List<double> data)//qrt
{
```

```
    int size = (int)Math.Sqrt(data.Count);
    double[,] matrixA = new double[size, size];
    double[,] matrixB = new double[size, size];
    List<double> result = new List<double>();

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            matrixA[i, j] = data[i * size + j];
        }
    }

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            for (int k = 0; k < size; k++)
            {
                matrixB[i, j] += matrixA[i, k] * matrixA[k, j];
            }
            result.Add(matrixB[i, j]);
        }
    }

    return result;
}

static List<double> MatrixProgram5(List<double> data)//summ3
{
    int size = (int)Math.Sqrt(data.Count / 3);
    double[,] matrixA = new double[size, size];
    double[,] matrixB = new double[size, size];
    double[,] matrixC = new double[size, size];
    double[,] matrixD = new double[size, size];
    List<double> result = new List<double>();

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            matrixA[i, j] = data[i * size + j];
            matrixB[i, j] = data[data.Count / 3 + i * size + j];
            matrixC[i, j] = data[2 * data.Count / 3 + i * size + j];
        }
    }

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            matrixD[i, j] = matrixA[i, j] + matrixB[i, j] + matrixC[i, j];
            result.Add(matrixD[i, j]);
        }
    }

    return result;
}

//public static ProgramN[] Programs = new ProgramN[] { //Gauss
```

```
    //    new ProgramN(GaussProgram1),
    //    new ProgramN(GaussReturnProgram1),
    //};
    public static ProgramN[] Programs = new ProgramN[] {

        new ProgramN(MatrixProgram1),
        new ProgramN(MatrixProgram2),
        new ProgramN(MatrixProgram3),
        new ProgramN(MatrixProgram4),
        new ProgramN(MatrixProgram5),

        new ProgramN(GaussProgram1), //Gauss
        new ProgramN(GaussReturnProgram1), //Gauss
    };

}
```

Лістинг 5. Головно вікна

```
class CreateRequestElement
{
    public Request CreatedRequest { get; set; }
    public int CountInputData { get; set; }
    public int CounterInputData { get; set; }

    public CreateRequestElement(Request CreatedRequest, int CountInputData, int
CounterInputData)
    {
        this.CreatedRequest = CreatedRequest;
        this.CountInputData = CountInputData;
        this.CounterInputData = CounterInputData;
    }
}

class MasterProcessor
{
    public List<Job> ListJobs { get; set; }
    public List<Data> ListData { get; set; }

    List<Data> ListAllDataPatern;
    public List<CreateRequestElement> ListRequests { get; set; }
    public Dictionary<int, double?> ListResults { get; set; }
    public List<SlaveProcessor> ListSlaveProcessors { get; set; }

    public int GlobalTime;

    public MasterProcessor()
    {
        GlobalTime = 0;
        ListJobs = new List<Job>();
        ListData = new List<Data>();
        ListRequests = new List<CreateRequestElement>();
        ListResults = new Dictionary<int, double?>();
        ListResults.Add(1, null);
        ListSlaveProcessors = new List<SlaveProcessor>()
        {
            new SlaveProcessor(),
            new SlaveProcessor(),
            new SlaveProcessor(),
            new SlaveProcessor()
        };
    }

    public bool InsertNewJob(Job newJob)
    {
        ListJobs.Add(newJob);

        int availableDataCount = 0;
        List<int> listAvailableData = new List<int>();
        foreach (int dataIdentifier in newJob.ListInputDataIdentifiers)
        {
            Data availableData = ListData.FirstOrDefault(d => d.Name ==
dataIdentifier && d.DataAddress != -1);
            if (availableData != null)
            {
                availableDataCount += availableData.CountData;
```

```
                listAvailableData.Add(availableData.DataAddress);
            }
        }

        ListRequests.Add(
            new CreateRequestElement
                (new Request(newJob.TaskIdentifier,
newJob.ListInputDataIdentifiers, listAvailableData,
newJob.ListOutputDataIdentifiers, newJob.Time),
                newJob.CountInputData,
                availableDataCount));
        return true;
    }

    public bool InsertData(Data newData)
    {
        ListData.Add(newData);
        foreach(CreateRequestElement elem in ListRequests)
        {

if(elem.CreatedRequest.ListInputDataIdentifiers.Contains(newData.Name))
            {
                elem.CreatedRequest.ListDataAddresses.Add(newData.DataAddress);
                elem.CounterInputData += newData.CountData;
            }
        }
        return true;
    }

    public bool RunTakt()
    {
        GlobalTime++;
        CreateRequestElement activationRequest = null;
        foreach (CreateRequestElement elem in ListRequests)
        {
            if (elem.CounterInputData == elem.CountInputData)
            {
                activationRequest = elem;
                break;
            }
        }

        if(activationRequest!=null)
            ActivationRequest(activationRequest);

        foreach (SlaveProcessor proc in ListSlaveProcessors)
        {
            proc.RunTakt();
            if(proc.FlagSloveTask)
            {
                proc.FlagSloveTask = false;
                ReadDataForSlave(proc);
            }
            if(proc.CurrentTakt == proc.TimeLoadData && proc.FlagRunTask)
            {
                WriteDataToSlave(proc);
            }
        }

        return true;
    }
```

```
    public bool ActivationRequest(CreateRequestElement ActivationedRequest)
    {
        SlaveProcessor priority_max = null;
        foreach (SlaveProcessor proc in ListSlaveProcessors)
        {
            if (proc.InputRegistr == null)
            {
                if (priority_max == null || priority_max.priority < proc.priority)
                    priority_max = proc;

                //proc.InputRegistr = ActivationedRequest.CreatedRequest;
                //proc.TimeLoadData = ActivationedRequest.CountInputData;
                ////priority_max.TimeLoadData = 0; slave set this
                //ListRequests.Remove(ActivationedRequest);
                //break;
            }
        }

        if (priority_max != null)
        {
            priority_max.InputRegistr = ActivationedRequest.CreatedRequest;
            priority_max.TimeLoadData = ActivationedRequest.CountInputData;
            //priority_max.TimeLoadData = 0;
            ListRequests.Remove(ActivationedRequest);
        }
        return true;
    }

    public bool ReadDataForSlave(SlaveProcessor proc)
    {
        for (int i = 0; i < proc.ListOutputData.Count; i++)
        {
            Data Paterndata = ListAllDataPatern.FirstOrDefault(d => d.Name ==
proc.ListOutputData.ElementAt(i).Key);

            Data data = new Data(Paterndata.Name, Paterndata.ArcIndex,
Paterndata.CountData, Paterndata.DataAddress);

            int address = ListResults.Keys.Max();
            ListResults.Add(++address, null);
            data.DataAddress = address + 1;
            foreach (double val in proc.ListOutputData.ElementAt(i).Value)
            {
                ListResults.Add(++address, val);
            }

            InsertData(data);
        }
        return true;
    }

    public bool WriteDataToSlave(SlaveProcessor proc)
    {
        //foreach(int address in proc.InputRegistr.ListDataAddresses)
        //{
        //    for(int i = address; i < ListResults.Count + 1; i++)
        //    {
        //        if (ListResults[i] != null)
        //        {
        //            proc.ListInputData.Add(i, ListResults[i]);
```

```
        //            ListResults[i] = null;
        //        }
        //        else
        //            break;
        //    }
        //}
        foreach (int id in proc.InputRegistr.ListInputDataIdentifiers)
        {
            for (int i = ListData.First(d => d.Name == id).DataAddress; i <
ListResults.Count + 1; i++)
            {
                if (ListResults[i] != null)
                {
                    proc.ListInputData.Add(i, ListResults[i]);
                    //ListResults[i] = null;
                }
                else
                    break;
            }
        }
        return true;
    }

    public bool AddSlave()
    {
        if (ListSlaveProcessors.Count < 8)
            ListSlaveProcessors.Add(new SlaveProcessor() { GeneralTimeHasTask = 0
});
        return true;
    }

    public bool RemoveSlave()
    {
        if(ListSlaveProcessors.Count > 1)
            ListSlaveProcessors.RemoveAt(ListSlaveProcessors.Count - 1);
        return true;
    }
}
```